A SYSTEMATIC FRAMEWORK TO ANALYZE THE DESIGN SPACE OF DNN ACCELERATORS

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF ELECTRICAL ENGINEERING
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Xuan Yang
October 2019

Abstract

Deep neural networks (DNNs) have been widely used to solve many modern machine intelligence problems. However, their outstanding accuracy comes at the cost of high computation complexity which limits their speed on conventional CPUs. This difficulty has encouraged researchers to create efficient DNN accelerators. Interestingly these designs use a variety of approaches, and have not converged over time. While each design states the advantages of its approach, without a comprehensive understanding of the global space, it is difficult to understand which design choices really matter. To address this issue, this thesis shows how the space of DNN hardware accelerators can be represented as scheduling choices, including computation orders, storage orders, etc.

Different DNN micro-architectures and mappings represent specific choices of loop order and hardware parallelism for computing the seven nested loops of DNNs. This observation enables one to create a formal taxonomy of all existing dense DNN accelerators, and systematically analyze the design space, including dataflow choice. The loop transformations needed to create these hardware variants can be precisely and concisely represented by Halides scheduling language.

By modifying the Halide compiler to generate hardware, we create a system that can fairly compare these prior DNN accelerators, and show that many different dataflows yield similar energy efficiency with good performance. As long as properly choosing the memory sizes to accommodate the efficient blocking strategy, it can ensure that most data references stay on-chip with good locality and the function units have high resource utilization. Thus, the hardware dataflow choices become less critical, but how resources are allocated, especially in the memory system, has a large impact on energy and performance. By optimizing hardware resource allocation while keeping throughput constant, we achieve up to 4.2× energy improvement for Convolutional Neural Networks (CNNs), 1.6× and 1.8× improvement for Long Short-Term Memories (LSTMs) and multi-layer perceptrons (MLPs), respectively.

Acknowledgments

In my life time, I have received help from many people. It is their help that provides me a positive attitude towards life and world. Among all these people, the help from some are particularly meaningful to me, and I wish to express my sincere appreciation to those who have guided and supported my during my Ph.D study.

First of all, I would like to think my advisor Mark Horowitz, who offers me the opportunity to perform research study in such a wonderful group, even when I am an amateur computer architecture researcher. Throughout all these years, in addition to being enlightened by his deep vision in the computer architecture field, he has also become the role model to my life. Mark has continuously provided me patient guidance in developing the insights in research problems, as well as the opportunities and supports in exploring my true research interests. His broad knowledge of many other research fields and deep understandings of humanities and society, have constantly impressed me and motivated me to stay curious and studious in my future life.

Many thanks to Christos Kozyrakis, Kayvon Fatahalian, Priyanka Raina and Boris Murmann for serving on my defense committee, and providing advice and guidance for this work in addition to my defense. Without the wonderful suggestions from them, it would have been difficult for this thesis to be completed and clearly presented. I would also like to thank Kayvon, Jonathan Ragan-Kelley and Andrew Adams, whose profound thinking of the hardware acceleration for both image processing and DNNs, and expertise in Halide have helped me in many aspects for this work. Additionally, I would like to thank Christos and Kayvon for the significant time they invested that helped me improve the quality of this dissertation.

Also I would like to express my appreciation to all the members of Stanford VLSI research group. Particularly, I would like to thank Ofer Shacham and Steve Richardson, who were always helpful and kind mentors in the group, and gives me many useful suggestions. I would also like to express my special thanks to my coauthors of this work, Jing Pu, Mingyu Gao, Jeff Setter, Qiaoyi Liu, Ankita Nayak, Steven Bell, Heonjae Ha, who have helped on the development of the Halide hardware generation framework, test and measurement on the FPGA and ASIC targets. Without their contributions, much of the research presented here would not have existed. I would also like to thank Byong Chan Lim, from whom I learned the knowledge about mixed-signal and SRAM designs. Hertfelt thanks extend to Suyao, Kahye, Nikhil, Alex and the rest of group members, who were brothers and sisters to me in this amazing community.

Indeed all will not be possible without the unconditional love and support from my family. It is their encouragement that paves the path for me to pursue my life goal and makes the person who I am today. Words cannot express the feelings I have for them.

Finally, I would like to thank my husband Yuming Kuang, who graduated from Stanford as well. For years during our Ph.D studies, we share our thoughts and insights on all kinds of matters, provide shoulders for each other, and solve the difficulties in life together. I am extremely grateful to him for being my source of strength and inspiration for these unforgettable years.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Deep neural networks (DNNs) have recently displaced classical image processing and machine learning methods due to their state-of-the-art accuracy on many tasks, particularly in recognition, localization, and detection [42]. However, their outstanding accuracy comes at the cost of high computation complexity and large memory footprint. Recent DNN models can require 30k-600k operations per pixel, and up to 100s of MBytes for weight storage, which makes DNNs a challenging workload for general-purpose processors to meet both energy efficiency and performance requirements.

DNN algorithms generally have high locality and large amounts of parallelism, making it fertile area for specialized hardware acceleration. Thus, as the number of applications for DNNs grows, so have proposals for DNN accelerators. NeuFlow created a 2D systolic array for convolutional neural networks (CNNs) [27], where each processing element (PE) can communicate with its neighbors. The DianNao family were built around customized inner-product units [12, 16], utilizing a single level of memory. Eyeriss highlighted the importance of on-chip dataflow on energy efficiency and proposed a row-stationary heuristic [13, 15]. And Google's TPU used a simple yet efficient systolic dataflow on a large 2D processing array [48]. In addition to these ASIC accelerator designs, many accelerators have been built targeting FPGA systems. Zhang et al. [99] adopted loop blocking technique to optimize data reuse and explored various tile sizes to minimize off-chip bandwidth. To further reduce the off-chip communication on the FPGA systems, Alwani et al. [8] fused the computation of multiple layers, thereby eliminating writing the intermediate data back to main memory. These are just a few of the recent publications on DNN acceleration.

Despite of the high efficiency and throughput achieved by these specialized accelerators, their development required significant design effort and the knowledge of DNN algorithms and architectures. To improve the productivity of DNN accelerator design, previous researchers have created a number of domain specific language (DSL) systems, that can compile high-level algorithmic descriptions to hardware designs by incorporating domain knowledge. Caffeine [100] generates FPGA designs from high-level network descriptions by integrating the hardware compilation flow with

Caffe. To improve its energy efficiency and performance, the accelerators are constructed from the hand-optimized parametrized design libraries. Similarity, VTA [60], a hardware generation system incorporated in TVM [11], designs another configurable architecture template for DNNs, which can be programmed by their proposed ISA.

Interestingly, even after considerable research effort, the resulting DNN accelerator architectures remain quite diverse. Some of the differences can be explained by researchers focusing on different parts of the total problem (hardware architecture, memory system design, or application scheduling) but it is unlikely that this explains the current diversity of solutions. To help understand the space of DNN accelerators and the current situation better, we create a systematic taxonomy of the complete design space, and a framework for generating the hardware implementation of designs in this taxonomy. This framework allows us to resolve the conflicting published reports of different "optimal" parameters and make it possible to fairly compare different design points. We also build an optimization framework by creating simple analytical models for energy and performance, which allow us to explore this large design space.

1.1 Thesis Outline

In order to create efficient accelerators, it is necessary to understand the algorithm. Chapter 2 starts by expressing DNN algorithms using seven nested loops. Next, it introduces the existing accelerators, with their different approaches used to improve energy efficiency and performance. With the large number of DNN accelerators being built, previous researchers also developed hardware generation frameworks based on high-level synthesis (HLS) or domain-specific language (DSL) to improve the design productivity of creating DNN accelerators. We adopt one particular DSL among them, Halide, to build our DSL system, as it decouples the functionality specification of the application (the *algorithm*) from how it is mapped to the underlying architecture (the *schedule*), and provides compact and concise scheduling primitives.

Build on the work presented in Chapter 2, we break the enormous design choices into three sets of parameters — *dataflow*, *loop blocking*, and *resource allocation*. From the perspective, we construct a three-dimensional logical representation of the design space to enable a systematical and comprehensive analysis. We show the dimensions can be nicely and concisely expressed as how they transform (block, reorder, and parallelize) the loop nest using the loop transformation based formal taxonomy we propose.

This loop transformation based taxonomy naturally aligns with the concept of Halide scheduling language. In Chapter 4, we demonstrates that the loop transformations we use to specify the micro-architecture and dataflow choices of a DNN accelerator are almost a subset of the loop transformations and memory allocation primitives provided by Halide's scheduling language [74]. Therefore, we extend Halide, enabling it to create hardware accelerators with any possible DNN dataflow and storage hierarchy, to allow us fairly compare various design choices in the 3D design space. Chapter 4 then introduces the parameterized architectural templates we designed for this system, and the implementation of the new compiler for generating efficient hardware.

To speedup the exploration of the design space, we build an optimization framework by employing simple analytical models for energy and performance. Chapter 5 begins by describing these models, and then validates them against synthesized designs. It then describes the flow of the optimization framework that utilizes exhaustive search to investigate the impact of each parameters in the space to determine the optimal schedule and hardware resource allocation.

Given the diversity of research solutions, our results using the optimization framework show that, with proper loop blocking, the specific dataflow used in the design *does not* have a significant impact on either energy efficiency or overall throughput. For energy efficiency, there is enough locality in most convolutional layers in DNNs so as long as they are properly *blocked*, most data references occur locally. Thus many schedules work well. On the other hand, for the layers that do not have enough data reuse to exploit, e.g., fully-connected layers that are not batched together, the overall energy efficiency is dominated by off-chip main memory accesses, thus the on-chip dataflow still does not have a large impact. From a performance perspective, it may first appear that dataflow choices can lead to significant utilization differences of the computation resource. However, with the technique of *replication*, which will be discussed in Section 3.4, many dataflows can achieve reasonable resource utilization.

In fact, energy efficiency is more tightly tied to the design of the hierarchical memory system and how each level in this hierarchy is sized. Every operation such as a multiply-add (MAC) involves multiple register file (RF) accesses. Since the cost of each RF fetch is proportional to the RF size, it is most efficient to adopt a relatively small RF. The hierarchy depth also matters, since the size ratio between the adjacent memory levels needs to be in a certain range to balance the total energy cost of accessing data at each level in the memory hierarchy. Using these insights, we created an efficient optimizer for these types of Halide programs, which achieves up to 4.2×, 1.6×, and 1.8× energy improvement over the Eyeriss accelerator for various CNNs, LSTMs, and MLPs respectively.

This thesis makes the following contributions:

- Introduces a systematic approach to precisely and concisely describe the design space of DNN accelerators as schedules of loop transformations.
- Shows that both the micro-architectures and dataflow mappings for existing DNN accelerators can be expressed as schedules of a Halide program, and extends the Halide schedule language and the Halide compiler to produce different hardware designs in the space of dense DNN accelerators.
- Demonstrates that many dataflow patterns achieve similar energy efficiency and performance, as long as proper loop blocking schemes are chosen to extensively reuse data, and replication techniques are used to fully utilize hardware processing resources.

- Discovers that the choice of micro-architecture and memory size is more important than the choice of dataflow, and that optimizing the memory hierarchy achieves $1.8\times$ to $4.2\times$ energy improvements for CNNs, LSTMs, and MLPs.

Chapter 2

Background

Deep Neural Networks (DNNs) have emerged as a popular solution for computer intelligence problems, with their superior capability to tackle tasks ranging from object recognition and detection to natural language processing. Such unprecedented accuracy, however, comes at the cost of high computational complexity and large memory footprint, which poses both energy efficiency and performance challenges to the underlying hardware.

To meet the demands for energy efficiency and performance, significant efforts have been devoted to the development of specialized accelerators for DNNs. Even highly-parallel machines can be utilized to fulfil the throughput requirement, with the massive data involved in DNN computation, it is still difficult to reduce energy consumption, especially memory energy, as data movement can be more expensive than computation. After first reviewing the basic DNN computation, Section 2.2 reviews the hardware acceleration approaches by previous researchers during the last decade, focusing on work that develop memory systems or study algorithm schedules. These groups investigated the memory organization and which loops to parallel to optimize locality. Section 2.2 also discussed the efforts to provide flexible accelerators for a wide range of networks, among them, FPGA systems have been commonly employed. Section 2.3 introduces high-level synthesis (HLS) that can generate FPGA implementations from high-level algorithmic descriptions. However, to create efficient accelerators with HLS, the knowledge of algorithm and hardware architecture is still indispensable.

To eliminate the need to understand the underlying architecture requires embedding this knowledge into the hardware generation tool flow. Such knowledge can be automatically extracted if the input application domain is sufficiently restricted. Section 2.4 starts by describing the computation and memory access patterns of image processing and the line-buffered template for capturing locality. Then it discusses prior domain specific language (DSL) systems that generate efficient FPGA accelerators leveraging this template. Since DNNs differ on the computation characteristics, Section 2.4 also presents another template used for DNNs — a double-buffered template, and reviews previous DSL systems that generate DNN accelerators based on this template from high-level languages.

Algorithm 1 CONV layer: simple seven nested loops.

```
for $b = 0$ to $B - 1$ do
  for $k = 0$ to $K - 1$ do
    for $c = 0$ to $C - 1$ do
      for $y = 0$ to $Y - 1$ do
        for $x = 0$ to $X - 1$ do
          for $f_y = 0$ to $F_Y - 1$ do
            for $f_x = 0$ to $F_X - 1$ do
              $\mathbf{O}[b][k][y][x] \mathrel{+}= \mathbf{I}[b][c][y+f_y][x+f_x]$
                          $\times \mathbf{W}[k][c][f_y][f_x]$
```

Noticing the computation commonality between image processing and DNNs, we use an existing image processing language, Halide, to describe our DNN applications. With the capability of decoupling schedules from algorithms, and the compact schedule primitives, the extended Halide system allows us to explore the choices of memory systems and schedules to generate efficient DNN accelerators.

2.1 DNN Algorithms

Advances in training deep, multi-layer networks have led to a resurgence of their use in many problem domains [42, 83, 89, 39]. In computer vision, deep neural networks (DNNs), such as convolutional neural networks (CNNs), multiple layer perceptrons (MLPs), long short term memories (LSTMs), have recently displaced classical image processing and machine learning methods for state-of-the-art performance on many tasks, particularly in recognition, localization, and natural language processing.

Counter to the classical, freely-connected model commonly associated with the neural network metaphor, deep neural networks are characterized by a highly restricted structure in which the network is organized into a pipeline or DAG of “layers,” and most layers are defined to perform a non-linear transformations on their inputs. In vision problems, these layers can be thought of as producing and consuming images, with their neurons organized into a regular 3D grid of pixels (with image dimensions x and y, and c for color channels). From this perspective, a DNN can be more clearly thought of as a specialized class of image processing pipelines, rather than as a biological neural model. The most widely used operations in this pipeline—convolution, local response normalization, pooling, and fully connected layers—correspond to the different “layers” used in the network.

- A *convolutional layer* (CONV) corresponds to a filter bank. In the standard case of 3D input and output, a convolutional layer computation is summarized as:

$$\mathbf{O}[b][k][x][y] = \sum_{c=0}^{C-1} \sum_{f_y=0}^{F_Y-1} \sum_{f_x=0}^{F_X-1} \mathbf{I}[b][c][x+f_x][y+f_y] \times \mathbf{W}[k][c][f_x][f_y]$$

and also shown in Algorithm 1 as seven levels of nested loops. The nested loops generate output feature maps (*fmaps*) $\mathbf{O}$, which have K channels of $X \times Y$ images, by processing the input fmaps $\mathbf{I}$ of C channels. The fmap data are processed in *batches* (B) to increase parallelism and data reuse. $\mathbf{W}$ contains the weights as K shift-invariant 3D stencil filters (one for each output channel) with size $C \times F_X \times F_Y$. Typically the dimensions of the kernels are much smaller than the image dimensions. By summarizing this computation as such nested loops, we can also express other layers or non-batched operations, by setting certain loop bounds to 1.

- A *local response normalization* (LRN) layer normalizes (scales) the value of each input by the sum of squared values in its neighborhood.

- A *pooling layer* performs a windowed reduction using some aggregation function (most commonly, *max*), decimating the input. This maps a $C \times X \times Y$ input to a $C \times X' \times Y'$ output, using a 2D stencil window of some size over the input within which the aggregation function is applied to produce a single output. Pooling and LRN layers have no learned parameters (weights).

- Finally, a *fully connected layer* (FC) is what is most commonly thought of within the neural network metaphor: an M to N mapping where all M inputs drive all N outputs, with unique weights for every input/output pair. This corresponds to an $M \times N$ matrix-vector multiplication, and with $M \times N$ unique weights has far more weight data relative to the size of the layer inputs and outputs ($O(\textit{input} \times \textit{output})$) than a convolutional layer. The computation of a FC layer can also be thought of as 1×1 CONV layer, which can be expressed using the same nested loops in Algorithm 1 with only C, K, and B loops, while other loops bounds are all 1.

These layers are normally connected as a linear pipeline to compose a CNN, as the example demonstrated in Figure 2.1. The output of each layer may be fed through a nonlinear *activation function*, such as a rectified linear unit (ReLU), then serve as the input to the next downstream layer. The CONV layers, which often appear at the beginning of the pipeline, are used to learn features from the input images. These extracted features are passed to the following FC layers to perform classification or prediction.

Most of the computational work in real CNNs, and most of the intermediate fmap data bandwidth, is in the convolutional layers. Meanwhile, the fully-connected layers, most commonly used in MLPs, LSTMs or at the end of a CNN pipeline, perform more work and load more parameters

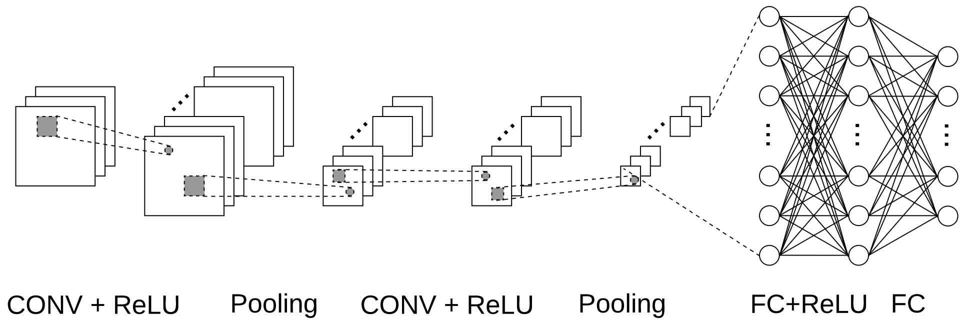


Figure 2.1: An example of CNN structure. Layers are connected as a linear pipeline.

per input or output. The nonlinear *activation functions* are typically local point-wise arithmetic operations which can be easily computed, they only have a small influence on computation cost and do not affect communication or locality at all.

During this decade, most widely used CNNs for vision applications range from the order of ten layers to hundreds. The AlexNet architecture [52] has five CONV layers with window sizes of 11×11, 5×5 and 3×3 interleaved with several local response normalization layers, pooling layers, and followed by two fully-connected layers. The VGGNet architecture [83] comprises several different network substructures, each composed of many CONV layers with 3×3 filter windows, interleaved with pooling layers, and followed by two FC layers. GoogLeNet [89] is constructed by repeating *Inception* module spatially and interleaving them with pooling layers, where each inception module is a combination of 1×1, 3×3, 5×5 CONV layers and pooling layers, with their outputs concatenated into a single output vector forming the input of the next stage.

To reduce the computation requirements and memory footprint, recent CNNs tend to avoid using FC layers, instead stacking more CONV layers to build deeper pipelines, i.e. ResNet [39], or design more efficient CONV layer variations. One of those examples—MobileNet [44] proposes depthwise separable convolution layer, which decomposes a single standard CONV layer into a depthwise convolution and a pointwise convolution, where the former one performs 2D convolution operation for each fmap with no accumulation at the end, the latter one is essentially a 1×1 convolution. We focus our evaluation on these newer networks, as well as the suite of applications demonstrated on recently published CNN hardware [12, 16]. Table 2.1 shows the computation and memory breakdown for AlexNet, VGGNet, GoogleNet and MobileNet architecture. From the table we can see that CONV layers are the most computationally intensive layer while FC layers, if adopted, consume the most memory.

Looking beyond vision domain, natural language processing is another fertile area for DNNs, particularly MLPs and LSTMs. MLPs is a stack of FC layers, each followed by nonlinear activation function [77]. The FC layer size ranges from to 10^2 to 10^7. LSTMs [43] is DAGs of LSTM units with

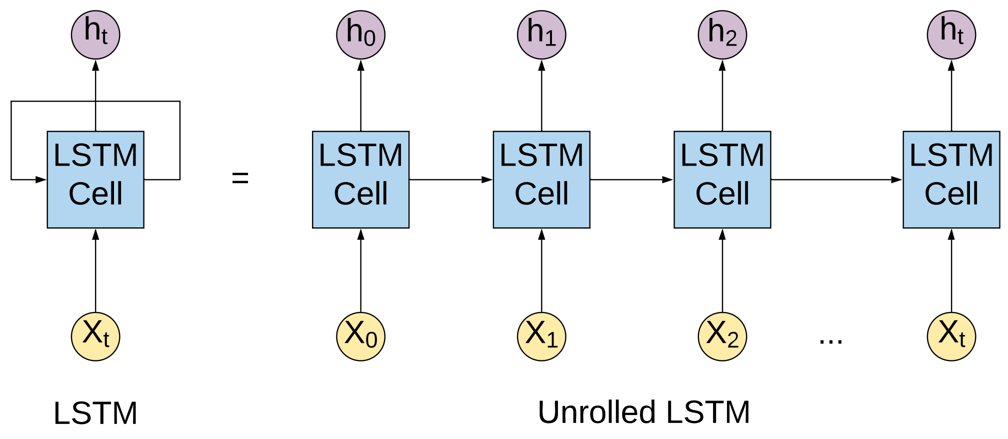


Figure 2.2: An example of LSTM structure. The LSTM cell looks at input X_t and outputs h_t. A loop allows information to be passed from one timestep t of the network to the next timestep $t+1$, equivalent to the unrolled network.

feedback connections, as presented in Figure 2.2. Each unit is commonly composed of a cell, an input gate, an output gate and a forget gate. The cell memorizes information over arbitrary timestamps, and the gates collaborate to regulate the information flow throughout the cell. Each gate performs a matrix-vector multiplication in non-batched case, or matrix-matrix multiplication in batched case, which can be regarded as FC layers due to the same computation performed (The batched one is obviously more energy and throughput efficient when latency allows). Therefore, during the inference phase, the feedback connection doesn't affect the characteristics of the computation, the required computation of LSTMs is essentially equivalent to MLPs, except the state matrices are shared by the LSTM units across different time steps. Such neural network with feedback loops were developed to improve the capability of utilizing the information from many steps back, and processing arbitrary long sequences for classifying or making predictions.

	MACs$\times 10^9$	**Mem (MB)**
AlexNet Convs	1.9	2
VGGNet16 Convs	15.2	59
GoogleNet Convs	1.5	23
MobileNet Convs	0.6	3
AlexNet FCs	0.058	234
VGGNet16 FCs	0.124	495
GoogleNet FCs	0.001	4
MobileNet FCs	0.001	1

Table 2.1: Computation (measured in number of multiply and accumulate operations) and memory consumption breakdown of state-of-the-art networks (each pixel and coefficient is 16 bits).

Even though modern DNNs include various layer types such as pooling, normalization, and element-wise operations, we focus on CONV and FC layers in this work since they dominate the computation and memory bandwidth.

2.2 DNN Accelerators

The superior accuracy of DNNs for many tasks, and the high computation complexity of the computation, has led to an increasing interest in providing acceleration of the DNN computation. While often independent research efforts converge to a few common approaches, this does not seem to be the case for DNN acceleration. These approaches differ on the way of organizing memory hierarchy, the schedule of the algorithm, and the hardware micro-architecture.

Early attempts for accelerating NNs can be traced back to 90s (Intel ETANN [2]), which were designed for accelerating shallow neural nets. The blossoming of DNN-based applications opened the floodgate of the DNN accelerator research, including Neuflow [27], Diannao [12], etc. Overtime, the hardware accelerations for DNNs tend to develop more complex memory systems to optimize locality. The NeuFlow architecture started with a 2D systolic array for CNNs, where each processing element (PE) communicated only with its neighbors, and data were streamed to and from DRAM [27]. To improve locality, its successor TeraDeep used a fixed loop-blocking strategy for CONV layers [31]. The DianNao family were built around customized inner-product units. The first generation used a single level of small buffers [12], while in a later iteration the original unit was surrounded by a large eDRAM that stored the complete data sets [16]. Another version specially built for embedded systems further extended to a 2D PE mesh that supported optimized inter-PE data propagation [26]. To further alleviate memory energy issues, another widely studied approach is near-data processing, which moves accelerator closer to DRAM. Neurocube combined spatial PE arrays with 3D-stacked DRAM to reduce the main memory access cost [50], QUEST designed an log-quantized datapath with stacked SRAMs [91], Schuiki et al. designed a near-memory accelerator along with RISC-V cores to perform both training and inference [79].

Another set of prior research focused on the schedule of the DNN algorithm, particularly which loops to parallelize, which is also called the hardware *dataflow*. With customized hardware using a spatial architecture, more recently, Eyeriss highlighted the importance of such on-chip dataflow for energy efficiency, and proposed using a row-stationary dataflow as a heuristic solution to maximize the data reuse of all types of data blocks [13]. Its follow-up work proposed to use a *replication* technique to further improve the throughput by increasing the computation resource utilization [15]. FlexFlow leveraged the complementary effects of different dataflow styles and mixed them on the same PE array to improve resource utilization [57]. Other prior work has also implemented architectures that are flexible to support multiple different dataflow types [54, 92, 20]. MAERI designed a configurable interconnection network within the accelerator to allow mapping DNNs using different dataflow types [54]. Wei et al. created an automated compilation framework to generate an efficient systolic array mapped on FPGA, based on their proposed analytical model for performance and resource utilization of different dataflows [92]. Its follow-up work, PolySA, leverages the polyhedral model to fully explore the dataflow space [20]. Tangram investigates the dataflow optimizations for coarse-grain parallelism, and proposes *buffer sharing dataflow* to eliminate excessive data duplication in the on-chip buffers for tiled NN accelerators [30].

Some groups extended this work to jointly study near-data processing with schedules/dataflows on the spatial PE arrays [29, 9]. Tetris [29] takes advantage of the high throughput and low energy characteristics of 3D memory to simplify dataflow scheduling for NN computations in near-data processing systems. NeuroStream proposed a flexible tiling mechanism along with a scalable computation paradigm to improve the overall resource utilization and throughput [9].

As MLPs, RNNs/LSTMs, GANs and other networks with potentially even higher compute complexity become pervasive in vision, sequence modeling and other fields, the hardware acceleration of these algorithms has drawn attentions from researchers. What makes their hardware acceleration more difficult is the complex network structures, for instance, LSTMs, Generative Adversarial Networks (GANs) etc. contain recursion and branches, thus are more similar to DAGs, instead of linear pipeline of layers. Google's TPU used a simple systolic dataflow on a large 2D array of PEs, which could also be used for MLPs and LSTMs in addition to CNNs, and adopted roofline model to improve the server latency [48].

Another common approach to create accelerators supporting a wide range of networks is to utilize FPGA systems. To accelerate CNNs, Zhang et al. [99] adopted the Roofline model to explore loop blocking, but considered only two levels of memory and only minimized off-chip bandwidth rather than total memory energy. Alwani et al. [8] fused the computation of different NN layers, and Li et al. [56] mapped the entire CNN onto an FPGA in a pipelined manner, both to reduce intermediate data writeback. Shen et al. [81, 82] optimized FPGA resource utilization by using a heterogeneous design. Sharma et al. [80] provided hand-optimized templates for generating dataflow architectures. Other previous attempts [33, 103, 66] utilized FPGAs to improve the energy efficiency of RNNs/LSTM accelerators, compared with CPU/GPU. DeltaRNN [28] implemented an RNN delta network update approach to reduce memory access with negligible accuracy loss on FPGA system. In terms of GANs, Song et al. proposed to reorganize dataflows to improve the data reuse for non-standard convolutional layers in GANs [85]. Similarly, FlexiGAN [96] and GANAX [97] utilized an architecture template that combines SIMD and MIMD models to alleviate the resource under-utilization issues for those non-standard convolutional layers.

Prior researchers also investigated various software-hardware codesign approaches for optimizing these accelerators. Some work focused on cooptmizing the algorithms with the hardware accelerators. The algorithm optimizations include but are not limited to reducing the model precision by quantization, compressing the model size using pruning, weight sharing, etc., designing compact network architecture, and transforming the computation into other efficient forms. While general processors usually support 8-, 16- and 32-b operations, the precision of activation and weights can be reduced to the range between 4 and 9 b for AlexNet across different layers with negligible impact on accuracy [49, 59]. Other work quantize the networks more aggressively into binary networks, such as [21, 22, 75]. Prior work that utilized quantization technique to reduce the precision of networks improve the performance and energy efficiency of both computation and memory [59, 65, 104, 24]. To further reduce the data movements and computation, some designs have explored DNN sparsity and proposed specialized dataflow schedules [38, 6, 102, 68, 98]. Another group of designs have transformed DNN processing into the frequency domain to reduce computations [101, 51] or log space to reduce computation cost [37, 58, 105]. Furthermore, other attempts have investigated and designed hardware-friendly neural network architectures to reduce network parameters and/or computation, including SqueezeNet [46], MobileNet [44].

2.3 High-Level Synthesis

In this large exploration of DNN accelerators, many researchers have devoted significant efforts on developing tools to reduce the design effort needed to create these DNN or other accelerators. One of the most promising approaches was High-level synthesis (HLS). Similar to designing efficient DNN accelerators, the major focus of these tools is determining the right schedule, particularly the data orchestration among the memory hierarchy, since memory system is the most critical component of DNN accelerator, as mentioned in Section 2.2.

HLS translates an algorithmic description written in untimed high-level languages like C/SystemC/Java, into a fully timed register-transfer level (RTL) implementation. In such a way, it can decouple the high level specification of the program from the low level design choices, including register allocation, clock-level timing, and pipelining. Many people hoped that this would enable application experts to create hardware, but as we will explain later in this section, this is not the case. This decoupling does raise the hardware design to the abstraction level, facilitating and accelerating the system-level exploration for architectural design space exploration. An additional advantage is that, the C-based behaviour simulations of HLS designs takes much less time than cycle-accurate simulation in HDL, which accelerates system level validation.

A HLS compiler first translates the high level description into a formal representation, with certain code optimizations including dead-code elimination, constant folding, loop transformation and so on. Then using user constraints on either performance, chip area or both, it performs hardware allocation, which determines the type and the number of hardware resources that the machine will use. Next, HLS schedules all operations required in the C specification into cycles onto the hardware and computes the number of cycles required. Depending on the user constraints, operations can be scheduled in several different ways: within one cycle, over multiple cycles, chained or executed in parallel. After scheduling phase, a binding step is executed to map each variable that holds values across cycles to a storage unit. Variables with non-overlapping or mutually exclusive lifetimes can use the the same storage units. Lastly, all decisions made in the previous tasks are applied to generate a synthesizable RTL design [23].

Starting from the late 80's, many HLS tools have been proposed and built. These tools were motivated by the very successful introduction and widely usage of the hardware description languages Verilog/VHDL. In the early 80s, SA-C translates single-assignment C into non-recursive data flow graphs to generate VHDL designs for image processing applications [25, 64]. ROCCC compiler focuses on the data path generation and optimizes the on-chip storage into "smart buffers" to capture the data reuse inside loop structures for high computational density applications [34]. AutoPilot/xPilot leverages the specific system platform information to optimize logic, interconnects, and generate efficient RTL code from C/ C++/SystemC descriptions for better performance and power [35, 19]. LegUp and Warp explore the hardware software codesign using application profiling to detect the critical kernels, and dynamically and transparently compiling programs into a heterogeneous system with FPGA and a processor. On the commercial side, available HLS tools provided by FPGA and EDA vendors include but are not limited to Vivado HLS [94], Catapult HLS [32], MaxCompiler [90] and Altera OpenCL [7].

Although the adoption of HLS significantly reduces FPGA development complexity and cycles by raising up the hardware design level, creating efficient FPGA design using HLS requires C-code built for hardware synthesis, which is quite different from developing high performance C applications. HLS facilitates resource allocation, state machine generation, stage pipeline and many tasks, thus makes it easy to explore schedules. However, in order to schedule programs efficiently, it is usually crucial to block the algorithm properly, and map the programs data to carefully partitioned memories to optimize data locality. Properly blocked data enables the machine to exploit locality and parallelism: smaller data blocks can fit in smaller and cheaper memories, enabling most data to be served there with smaller cost. Such efficient data blocking and program mapping requires understanding and analysis of the memory access pattern and the underlying hardware features, which we will demonstrate more in Section 3. This issue can become more challenging when multi-level memory hierarchy is utilized to further optimize locality and performance.

This knowledge is also indispensable for HLS programmers to build efficient hardware, for instance, instantiating the proper local memory structures to optimally capture locality based on the memory access pattern of different applications. To create these systems, it requires completely distinguished C code from traditional C code that that runs application on general processors. Therefore, generating high performance HLS code requires knowledge of both algorithm and architecture. So when applied well, it can make hardware designs more efficient, releasing application experts from the burden of creating hardware.

Another major drawback of HLS is the lack of readability and portability. When architecture is mixed with the algorithm knowledge, designs can produce high performance implementations, but like highly tuned C-code, it is generally difficult to understand. It is often hard to take this code and modify it. These limitations are due to the fact that scheduling optimizations are blended with the functionalities within HLS code. These issues add to some of the limitations of current HLS tools. Also we have found HLS tools sometimes lack robustness when analyzing and compiling loop nests with complex schedules. We will provide some examples and more details in the next section to present the issues when creating DNN accelerators using HLS.

To free the designer from knowing about architectural issues requires making the tool incorporate this knowledge. One common approach to accomplish this goal is to restrict the application domain, thus limiting the knowledge that the tool needs to operate. This domain restriction makes it possible to analyze the schedule and mapping for an program to achieve optimal locality and parallelism, when compiling onto hardware. The benefit of domain knowledge, together with the motivation to improve robustness, readability and portability upon HLS, have led to strong interests in developing DSL systems.

2.4 DSL Systems

When focused on one specific domain, the tool can contain the proper microarchitectural templates and use them to automatically and efficiently generate high performance implementations from a domain specific language. With knowing application characteristics, a DSL compiler can organize the memory structures, and block the algorithms accordingly to map onto the optimized data storage for better locality. For example, image processing kernels generates each output pixel using a window of input pixels, and the window moves in raster-scan order. With such a compute pattern, a *line-buffered* pipeline is commonly utilized by image processing DSL systems to capture the data reuse across overlapped windows. For DNN systems that move blocks of data between units, *double buffers* are commonly employed to overlap data communication between computation units of the next data block with the ongoing computation of the current block. The next two sections will introduce DSL systems for these application areas, and describe the memory organizations in more detail.

2.4.1 Image Processing DSL Systems

Image processing applications have extreme locality and parallelism making them fertile area for hardware acceleration, and DSL systems for generating them. To minimize the number of DRAM accesses, image processing algorithms are typically expressed as deep pipelines, or more generally, directed acyclic graphs (DAGs) of computation kernels, where each kernel takes one or more images as input, and produces an intermediate output image which is feed to another kernel. Initially it might seem that the deep pipelines would require a large amount of memory for all the intermediate images. To minimize this storage, the system takes advantage of the fact that pixels generally flow

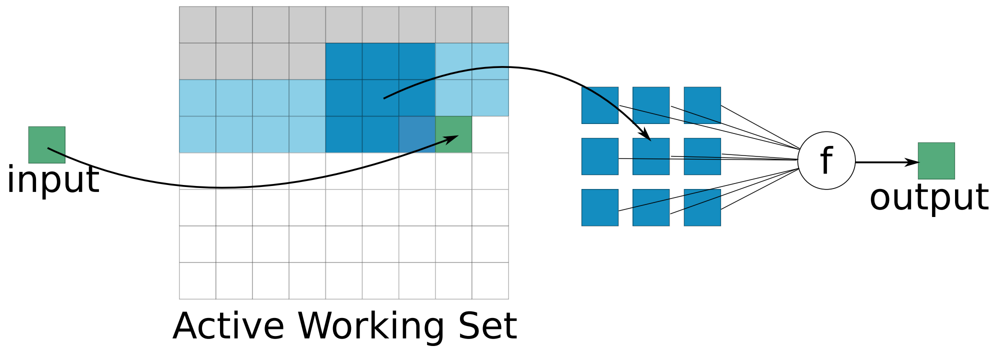


Figure 2.3: A line buffer (the blue box) captures the active working set for a stencil kernel using minimum storage.

through this pipeline in raster-scan order. In addition, most kernels in the DAGs are *stencil kernels*, e.g., blurring, sharpening, etc., which calculate each output pixel from a local stencil window of input pixels. With stencil kernels, all the data required to generate an output pixel can fit within a small memory block, thus the storage for the intermediate images can be *folded*, reducing it to a few lines of the image. For instance, given a 3×3 convolution kernel, if the input pixel rate from the producer kernel equals the output pixel rate, the minimum working set of the intermediate image is only two rows of pixels, as shown in Figure 2.3. These optimized storage units that capture the minimum required working sets are called *line buffers*.

A Line buffer is commonly placed between kernels in image processing pipelines, since it can maximize data reuse and minimize memory communication. With pixels continuously calculated, a line buffer can reuse the pixel from one stencil window to the next with minimal memory accesses. This memory structure also ensures all the intermediate images are buffered locally within a smaller and cheaper memories, and the global data fetching and write-back only occur once at the beginning and end of the pipeline respectively. Therefore, most custom hardware for image processing applications use line buffer pipeline as the microarchitectural template. This template separates kernel stages by line buffers and computes all kernel stages concurrently within the pipeline.

Of course it is possible to create line buffer and pipelines using HLS [72, 71], and these results indeed show that implementing a line buffer targeting a FPGA (using HLS) is more complex than targeting a CPU (using C language), as the hardware line buffer implementation requires explicit management of storage allocation and data transfer. Specifically, a software program for CPU can automatically exploit data reuse of adjacent stencils by caches and the register files of processors, thus only needs to organize the loop nests to minimize data reuse distance. A HLS line buffer implementation has to explicitly allocate the storage for stencil window, and carefully manage the data movement within and throughout the stencil structure.

To reduce the complexity of creating proper microarchitecture using HLS, many researchers have created DSL systems for image processing hardware generation. Those DSL systems leverage the microarchitectural knowledge of the domain to generate efficient implementation from high-level algorithmic specification. Specifically, Darkroom [40, 78], HIPAcc [76] and PolyMage [18] DSLs systems take image processing algorithms coded in a DSL, compile and map them onto FPGA or ASIC, using a line buffer pipeline microarchitectural template. To improve overall throughput, PolyMage splits images into tiles, and perform coarse-grain parallelism by processing each tile in parallel. Recent HIPAcc followup work [67] further performs fine-grain parallelism by vectorizing the whole pipeline and processing vectors of pixels. Both HIPAcc and PolyMage emit HLS code, and leverage Vivado HLS [94] to translate HLS code to RTL implementation.

The aforementioned systems all use line buffer pipeline to improve energy efficiency for image processing applications, with the ability to scale the design as a whole with coarse-grain parallelism to suit for a larger FPGA to achieve higher throughput. However, with different program sizes, it is crucial to support adjusting the throughput rate of each pipeline stage individually to achieve energy savings. Rigel [41] and HalideToHardware [71, 72] are the examples of variable-rate DSL systems that utilize a multi-rate line buffer pipeline to provide such flexibility. They allow developers to set the throughput rate to be greater or less than one pixel/cycle, which are realized by vectorization or time multiplexing respectively. Rigel embeds the rate setting into the high level program, by providing a set of multi-rate modules for explicit instantiation. HalideToHardware takes advantage of Halide's decoupling of schedule from algorithm, and uses the high-level schedule primitives to set the data throughput. After specifying the throughput using the primitives, the HalideToHardware compiler automatically lowers the high-level program into multi-rate pipeline mappings. This system makes exploring hardware design choices, particularly data throughout, easy by changing the program's schedule. Thus, like the CPU and GPU code it generates, generating optimized hardware doesn't need to change the algorithm code, it is done through the schedule. To leverage this benefit, we also use Halide as the high-level language for our DSL system, and extend the HalideToHardware system to support DNN applications in addition to the image processing domain.

2.4.2 DNN DSL Systems

Similar to imaging pipelines, DNNs are also composed as DAGs of multiple layers, and have greatly benefited from hardware acceleration, but require different memory structures to exploit their intrinsic parallelism and data locality.

Although prior attempts have made great studies in automatic accelerator generation in the image processing domain, the detailed computation and data reuse patterns of image stencil kernels and

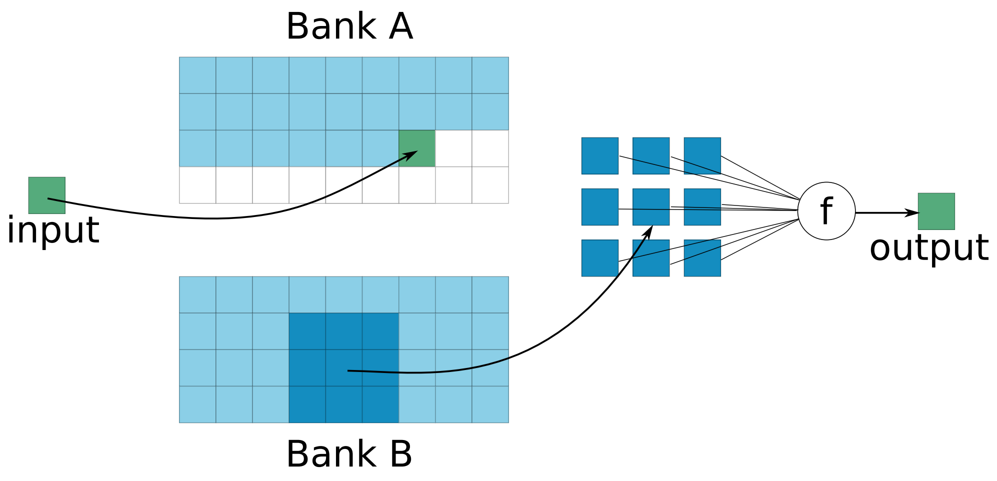


Figure 2.4: A double buffer uses two banks to hide data fetch latency.

DNN layers are significant different. A different optimal memory structure is needed to optimize locality. Unlike image processing, which utilizes a line buffer to capture the stencil reuse with minimum capacity, DNN layers may not have stencil reuse, such as FC and 1×1 CONV layers. Even for convolution layers which have this reuse, the large number of input and output channels associated with each kernel, changes the memory optimization problem. The large amount and large reuse of this data generally means one should build a multi-level memory hierarchy, and block the application to use it. Since we know exactly what data we will need to process next, highest efficiency is obtained by fetching the next data block while computing the results from the current data. *Double buffers* are frequently used to achieve such goal in DNN acceleration [99]. As depicted in Figure 2.4, it uses two banks that work in a ping-pong fashion, where one bank serves data for the current computation while the other bank is written with the next data block. Using such optimized storage units, the data communication latency can be hided under computation cycles, as long as there is sufficient computation to perform, which is usually the case for DNN applications. Therefore, similar to previous DNN accelerators [99, 13], we also adopt double buffer pipeline as our architectural template for DNN hardware generation.

```
alloc A[16]
alloc C[1]

for m = 0 to 16
  // Fetch and buffer one row of A for reuse across loop n
  for k = 0 to 16
    fetch A at [m, k], store in A[k]
  for n = 0 to 16
    // Compute an output element by dot product
    for k = 0 to 16
      C[0] += A[k] * B[k, n]
  write back C[0] to C at [m, n]  // Write back a complete output C element
```

Listing 2.1: Pseudo-code of matrix multiplication between two 16×16 matrices for CPU. CPU target allocate local buffers to optimize the locality for A and C matrices.

Despite of being popular for mapping DNNs onto FPGAs, using HLS for DNN hardware generation again requires the designer to have hardware design expertise to manually create the right memory structure and manage the data access. Also as mentioned in Section 2.3, HLS programming suffers from poor readability, portability and robustness. Listing 2.1 and 2.2 present the distinction between the pseudo-code of a matrix-multiplication, targeting at CPU and FPGA. Though both pieces of the code allocate local buffers to optimize the data locality, the FPGA code is more complex, as its local buffer structure is implemented as a double buffer, and the code has to explicitly manage the data assignment and communication. In Listing 2.1, each iteration of loop `m` computes one row of output C matrix by reusing each row of A matrix across all iterations of loop `n`, thus a local buffer is allocated to avoid the data refetching. Note the CPU code assumes that the data prefetching and reuse will be automatically exploited by caches and register files of the processors, but this assumption doesn't hold for FPGA target, thus explicit double buffer management is necessary. As shown in Listing 2.2, the FPGA implementation creates two banks for buffering A matrix (Line 1), and adopts the software pipelining technique to manually perform data prefetching to hide the data transfer latency under the computation (line 5 to 29).

After adding the complicated and explicit local memory management for generating efficient hardware, design space exploration becomes harder, since evaluating each choice likely requires different memory patterns, and significant HLS code change. It is inefficient and challenging using HLS to analyze the entire design space, especially for large problems. For the matrix multiplication example demonstrated in Listing 2.1 and 2.2, the space contains a large number of design points that break the matrices into smaller ones in various ways and buffer them accordingly to optimize the locality. As the spaces grow exponentially with the dimensions and sizes of the problems, the design space for DNNs can be even larger, and we will provide more detailed analysis about this in Chapter 3.

```
alloc A[2][16] // Allocate two banks for double buffering
alloc C[1]
init flag = 0  // Initiate iteration flag for counting the bank index

// Filling phase: prefetch the first row of A, store in bank A[0]
for k = 0 to 16
  prefetch A at [0, k], store in A[0][k]

// Steady phase: use double buffers to hide data fetching of each new row
for m = 0 to 15
  // Fetch and buffer one row of A for reuse across loop n
  for k = 0 to 16
    // The newly fetched row of A is stored in the other bank A[1-flag]
    fetch A at [m, k], store in A[1-flag][k]
  for n = 0 to 16
    // Compute an output element by dot product
    for k = 0 to 16
      // Consume the row of A stored in the current bank A[flag]
      C[0] += A[flag][k] * B[k, n]
    write back C[0] to C at [m, n]  // Write back a complete output C element
  // Update iteration flag, so that flag will be set to 1 if it is 0, and vice versa
  flag = 1 - flag

// Draining phase: compute the last row use the last row stored in bank A[1]
for n = 0 to 16
  // Compute an output element by dot product
  for k = 0 to 16
    C[0] += A[1][k] * B[k, n]
  write back C[0] to C at [15, n]  // Write back a complete output C element
```

Listing 2.2: Pseudo-code of matrix multiplication between two 16×16 matrices for FPGA. In addition to allocating local buffers to optimize locality for A and C matrices, the FPGA target also allocate a double buffer for A using two banks. To manage the two banks to work in the ping-pong fashion to hide the data fetching latency, FPGA target explicitly performs software pipelining to prefetch the next block of data while consuming the current block.

Similar to image processing acceleration, to automate the implementation of optimal microarchitectures and schedules, prior work created DSLs for DNNs. For instance, Caffeine [100] and DNNWeaver [80] can directly compile high-level network descriptions to a FPGA target by integrating with Caffe [47]. To achieve good performance, both of them develop hand-optimized design templates and utilize those parametrized templates as libraries when generating DNN accelerators. Similarly, VTA [60] is another programmable deep learning architecture template, integrated with TVM [11] (a deep learning compilation stack) to support divergent hardware backends. To provide the flexibility for diverse and evolving models, VTA proposes a two-level ISA and uses a just-in-time (JIT) compiler, along with a parametrized architecture. HeteroCL [55], another framework extended from TVM, consists of a Python-based domain-specific language (DSL) and a compilation flow targeting at FPGA. Inspired from Halide [74], it also provides a programming abstraction that decouples algorithm specification from hardware mapping choices, to achieve high performance. T2S-tensor [86], directly build on Halide, also leverages the decoupling feature to generate efficient systolic array implementations for dense tensor kernels on spatial architectures, such as FPGAs and CGRAs. Since Halide provides the benefits of decoupling algorithm from schedule, we employ and extend Halide as our DSL system for compiling DNNs onto hardware.

2.5 Halide Language

Halide [74] is a domain-specific language (DSL), that originally targeted at generating high-performance image processing implementations with readability, portability and modularity. The key idea in Halide is to split the computation to be performed (the *algorithm*) from the decisions about storage and the order in which it is done (the *schedule*).

Due to the decoupling between *algorithm* and *schedule*, the Halide compiler can translate the high-level, architecture-independent specification, defined in Halide algorithm, to low-level high-performance machine code for different machines and architectures, with proper schedules. In other words, with the same algorithm code, the schedules can be tuned (manually by programmer or automatically by auto-scheduler [61]) to significantly improve the performance, based on the characteristics of underlying machine. This improvement is not limited to the general backends such as CPU and GPU that are originally supported by Halide system. Pu et al. [72] also extend the system to describe and map image processing applications onto a CPU/FPGA heterogeneous systems, by taking advantages of the portability of Halide algorithms, and the compact scheduling language to organize the computation [72].

Even though Halide is originally designed for image processing, it is generally applicable to dense loop-structured computation including linear algebra and DNNs. This is because both matrices and feature maps can be regarded and stored as images, and the schedules of the computation essentially perform loop transformations. As a result, Halide provides a compact and elegant language to represent loop transformations. These transformations along with commands that create intermediate storage enables us to extend Halide further to generate hardware accelerators for DNNs in addition to image processors. The rest of this section presents more information about the Halide language and its programming model.

2.5.1 Halide Algorithms

Halide represents the computation algorithm in pure functional form. In a DNN algorithm, fmaps in a layer can be represented as Halide functions, defined over an infinite domain, which maps pixel coordinates to their values. Halide functions can be simple expressions, or reductions over a bounded domain. Reductions are particularly useful for expressing stencil computation in image processing and DNNs, as it can express iterative or recursive computations. They are typically composed of an initial definition and an update reduction function. The initial definition initializes each point in the function domain. While the reduction function redefines the value at each point by using update expressions, which can contain references to the same function. Unlike a pure function, such as the initial definition, reduction is not applied over an infinite domain, but rather a bounded one. This bounded domain that defines the order in which the reduction function is applied, is called `reduction domain`. The following example shows the Halide algorithm for a CONV layer with $3 \times 5 \times 5$ filter size.

```
1 // To perform a 5 x 5 convolution with 3 channels
2 // RDom(xMin, xExt, yMin, yExt, kMin, kExt)
3 RDom r(-2, 5, -2, 5, 0, 3);
4 output(x, y, k) = 0;
5 output(x, y, k) += input(x + r.x, y + r.y, r.z)
6                   * w(r.x + 2, r.y + 2, r.z, k);
```

The `RDom` keyword defines a multi-dimensional reduction domain, over which an iterative computation such as a summation is performed. A `RDom` is defined by the minimum position and extent in each of its dimensions. In the CONV layer example, the `RDom` covers the width and height of the filters and the number of input fmaps, over which the accumulation will iterate. The initial definition (line 4) uses a simple assignment to initialize each point in the `output` domain. Then the CONV filtering is performed using summation to update the value of `output` (line 5).

For reduction functions, the dimensions of the reduction domain can only be reordered or parallel if the update function is associative. As a practical simplification for applications such as image processing and DNNs, Halide model restricts the bounding region to be axis-aligned, so that the regions can be defined and analyzed using simple interval analysis. This facilitates the bound analysis and compiler inference of the storage for each function and loop with any expression constructed in Halide language.

As mentioned in Section 2.1, DNNs are organized as a pipeline or a DAG of layers. To build such pipeline or DAG, functions can be composed into graphs, in which the function definitions can refer to other previously defined function values. For instance, a two stage CNN can be expressed as a chain of four functions:

```
// To perform a 5 x 5 convolution with 3 channels
// RDom(xMin, xExt, yMin, yExt, kMin, kExt)
RDom r1(-2, 5, -2, 5, 0, 3);
conv1(x, y, k) += input(x + r1.x, y + r1.y, r1.z)
                  * w1(r1.x + 2, r1.y + 2, r1.z, k);
relu1(x, y, k) = select(conv1(x, y, k) < 0, 0, con1(x, y, k));

// To perform a 3 x 3 convolution with 64 channels
// RDom(xMin, xExt, yMin, yExt, kMin, kExt)
RDom r2(-1, 3, -1, 3, 0, 64);
conv2(x, y, k) += relu1(x + r2.x, y + r2.y, r2.z)
                  * w2(r2.x + 1, r2.y + 1, r2.z, k);
relu2(x, y, k) = select(conv2(x, y, k) < 0, 0, conv2(x, y, k));
```

Function `conv1` and `relu1` compose the first CONV layer, which performs 5×5 convolution to process 3 input feature maps (ifmaps) and generate 64 output feature maps (ofmaps). These 64 ofmaps serve as the inputs to the second CONV layer, composed of `conv2` and `relu2`. Reference to function `conv1` are used in the definition of function `relu1`, so does function `relu1` and function `conv2`, indicating the data dependency between adjacent two functions. Note, the region of each function to be computed is not given by the program, but instead derived from the compiler automatically from the realization bound of the output of the pipeline. Specifically, the compiler can analyze the computation pattern and dependency of each function, and determine the required region of the intermediate functions based on the final output size. For this example, if the ofmap `relu2` is set to 256×256, the required size of 256×256, 258×258 and 258×258 for function `conv2`, `relu1` and `conv1` will be computed by the compiler, respectively.

2.5.2 Halide Schedules

While the algorithm defines the functionality of the computation, it does not specify the ordering of parallel operations or data accesses. Generally, the number of possible execution orders of an image processing or DNN are numerous. Luckily, these algorithms can mostly be abstracted as graphs of functions over regular grids, leading to the optimal organization of computation on parallel machines being a regular structure. Therefore, leveraging this characteristic, Halide formulates this huge scheduling space, and proposes a scheduling language to specify the computation order and storage organization. Specifically, Halide schedules, which consist of *scheduling primitives* applied to various stages of the pipelines, addresses two issues:

- At what granularity to compute and store each function, and at what granularity to interleave this computation with other functions.
- In what order the domain of a function should be traversed for evaluation.

The Domain Order Halide scheduling model defines the traversal order in the required region of each function's domain. This order is named as *domain order*, which essentially use a traditional set of loop transformations, such as tiling, vectorization and unrolling. Because Halide's model of function is data parallel by construction, dimensions can be interleaved in any order, and traversed sequentially or in parallel using the loop transformations. In the language, these transformations can be specified using the following *scheduling primitives*.

- Each dimension can be traversed sequentially (by default) or in parallel (`f.parallel(x)`).
- Constant-size dimensions can be unrolled (`f.unroll(x)`) or vectorized (`f.vectorized(x)`).
- Dimensions can be reordered, for instance, `f.reorder(y, x)` moves the loop over y inside the loop x.
- Dimensions can be split, for instance, `f.split(x, xo, xi, factor)` split dimension x by a `factor`, creating an outer dimension `xo` and an inner dimension `xi`. The original reference is replaced by $xo \times factor + xi$.
- Combining loop splitting and reordering is loop tiling, which is a useful technique to exploit memory locality. Halide provides a syntactic suger `f.tile(x, y, xo, yo, xi, yi, xfactor, yfactor)` to conveniently perform loop tiling, which splits `x` by a factor of `xfactor` and split `y` by a factor of `yfactor`, then transposes the inner dimension of `y` with the outer dimension `x` to traverse over tiles.
- Two dimensions can be merged into a new dimension by fusing loops (`f.fuse(x, y)`).

By performing loop transformations on the dimensions of a function using above primitives, its domain can be traversed in any regular order, providing the opportunity to optimize locality and performance.

The Call Schedule Beyond specifying the order of evaluation within the domain of each function, a Halide schedule can also specify at which granularity the computation and storage of stages are interleaved within a pipeline. These choices are named the *call schedule*, while each function's call schedule is defined as the loop level of its callers where it is computed and stored for reuse. The two most commonly used schedule primitives are:

- `f.compute_at(g, d)` specifies the function `f` to be realized at each iteration of the loop over domain `d` of the function `g`. The region of `f` to be evaluated each time can be automatically inferred from Halide compiler based on the geometry of domain `d` and the data dependency between `f` and `g`, no programmer specification is needed.
- `f.store_at(g, d)` specifies the memory to be allocated at each iteration of the loop over domain `d` of the function `g` for storing and reusing values of the function `f`. The size of the allocation required each time can be automatically derived from Halide compiler in a similar way.

Together, the domain order and call schedule provide the capability to schedule stencil pipelines on rectangular grids. They allow the locality of a stencil pipeline to be optimized, using the loop transformations defined within the domain with the inter-stage interleaving schedules. However, to further exploit locality for applications that involve massive data and have extreme locality, a single level of buffer constructed from the function itself is insufficient. Therefore, Halide also provides another schedule primitive `in` to instantiate additional pieces of storage. Specifically, `f.in()` creates and returns a new identity function that wraps `f`, thereby replacing all calls to function `f` with the calls to the wrapper. Using this primitive, together with `compute_at`, enables moving a subset region of a function into a smaller local buffer, thus substituting the accesses to the original function storage with cheaper ones to the local buffer.

2.6 Summary

With DNN applications becoming pervasive in the last decade, the solutions for accelerating DNNs exploded. Many researchers have studied micro-architecture, schedule and software-hardware codesign for DNN acceleration. Due to the high computation complexity and large parallelism of DNNs, spatial architecture such as a systolic array has been widely adopted by prior work. The spatial architecture is commonly incorporated with double buffers to provide low data fetching latency and high performance. With memory energy from all levels in the memory hierarchy constantly dominating the overall system energy, prior researchers also investigated different approaches to optimize locality and energy efficiency. This includes how to efficiently schedule DNNs onto the hardware, particularly the data movement inside spatial array and across memory hierarchy.

When mapping functions onto the hardware, there exists a large number of scheduling choices to schedule the seven nested loops that express each CONV and FC layer, including loop tiling, unrolling, reordering, etc.. Composing various choices of tiling, and all possible granularities of store and compute, it leads to an enormous scheduling space. Fortunately, Halide provides sufficient primitives to express all these scheduling choices, thereby nicely and compactly describing the huge schedule space.

Searching for the optimal choice in this huge space is difficult, particularly with each choice making its trade-off between parallelism, locality and redundant computation. To facilitate searching for the most efficient mapping on hardware for DNNs, we leverage Halide language to build our DSL system. The Halide's scheduling language, built on top of the loop transformation concepts, provides a perfect model to describe the optimizations and cover the scheduling space for developers when mapping image processing onto CPUs and GPUs. The next chapter will propose using the loop transformation concepts to precisely and comprehensively describe the design space for CNNs. Inspired by these concepts, Chapter 4 will demonstrate how we can extend the scheduling model for expressing the hardware mapping and scheduling space for CNNs.

Chapter 3

The Design Space of DNN

Papers describing the DNN accelerators discussed in Section 2.2 have shown energy and performance improvements over general-purpose baselines, and explored how the parameters they experimented with affect performance and efficiency. Unfortunately, without an understanding of the global design space, each paper explores a different part of the space — perhaps coupling together independent parameters — so this approach leads to conflicting reports on the "optimal" parameters.

To avoid this problem, and help developers better understand the impact of each parameter, a systematical analysis of the space of all possible dense DNN accelerators is necessary. We start by looking at the subspace that have been explored by prior works. Section 3.1 reviews and summarizes the architecture templates and schedule strategies commonly used by previous DNN accelerators. With the spatial architecture commonly utilized, optimizing the schedules becomes crucial, particularly finding the proper loops to parallel, also named *dataflow*. Section 3.2 introduces previous efforts on systematically describing the schedule choices, including stationary-based, loop next-based approaches, etc.. These proposed taxonomies can facilitate the classification of existing DNN accelerators, and exploration of schedule choices.

Since all accelerators compute the same results, the differences among accelerators must be in the resources available to compute the result, and the way the computation is scheduled to use these resources. However, prior taxonomies usually purely focused on the schedule and mapping of the algorithms, and coupled schedule choices with the architecture design choices. To individually investigate each parameter and comprehensively analyze the space, in Section 3.3 we consider a 3D design space for DNN accelerators. To precisely represent the dataflow choices in the space, Section 3.2 proposes a formal taxonomy based on the spatial loop unrolling, as it perfectly aligns with the Halide schedule primitives.

3.1 DNN Accelerators

The computational complexity of DNN algorithms and the demand for high energy efficiency has lead to a surge in research on hardware accelerators. The computational model of DNNs are essentially DAGs of layers, with CONV and FC layers dominating the computation workload. The majority of the computation of those layers are sets of multiply-and-accumulate (MAC) operations, which can be easily parallelized. To improve the overall throughput, highly-parallel compute paradigms are widely exploited. The CPUs or GPUs are temporal architectures, since they execute instruction sequentially in time, whose parallelism can be improved by vectorization (SIMD), multi-threading (SIMT), etc. These machines feature great flexibility, but depend on a centralized control unit (instruction unit) to orchestrate the data movement among all Processing Elements (PEs). In other words, PEs don't have the capability to independently fetch data from the memory hierachy or other PEs. To overcome the control and communication overhead posed by this centralized control unit in the temporal architecture, spatial architectures has drawn a great attention from previous researchers as better alternatives for accelerating DNNs [13, 48, 99, 29]. They replace the single control unit with a dedicated control logic along with local buffers inside each PE. Such configuration enables both parallel processing and dataflow processing, which relay data from one PE to another to connect PEs as processing chains.

Spatial architectures are widely employed for both ASIC and FPGA platforms. For instance, Diannao designed a Neural Functional Unit (NFU), which utilizes a set of multipliers followed by a reduction tree to perform dot product in vectorized fashion [12]. With such NPU, their spatial architecture is able to optimize the computation energy efficiency and performance, however due to the inappropriate memory hierarchy configuration, the overall energy in this system is significantly dominated by the memory energy. This was an early result which indicated that optimizing memory energy is usually more critical than optimizing computation energy for DNN workloads. The reason is simple: performing one MAC operation requires three operand reads and one output write. Each access to the main memory is more expensive than computing one MAC operation. Therefore Diannao's follow-up work DaDiannao [16] allocated a large on-chip eDRAM, attempting to buffer all weights on-chip to alleviate the memory energy issue by reducing the off-chip memory fetches. While given the DNN statistics in Table 2.1 in Section 2.1, the required on-chip buffer size only to store all the weights is already on the order of Megabytes to hundreds of Megabytes, leading to expensive and slow data fetches from these buffers as well.

To optimize the memory energy, it is crucial to properly design the hardware architecture, particularly the memory hierarchy, and carefully orchestrate the data movements among different memory levels. Some other researchers have studied co-optimizing algorithms with hardware to reduce the model sizes, including pruning the models into sparse networks [38], or designing more hardware-friendly networks such as squeezenet [46] and mobilenet [44] as mentioned in Chapter 2. This work, focuses on the scheduling/mapping and architecture design for existing dense networks, so the next section discusses more details about previous proposed DNN architecture for these applications.

3.1.1 DNN Accelerator Architectures

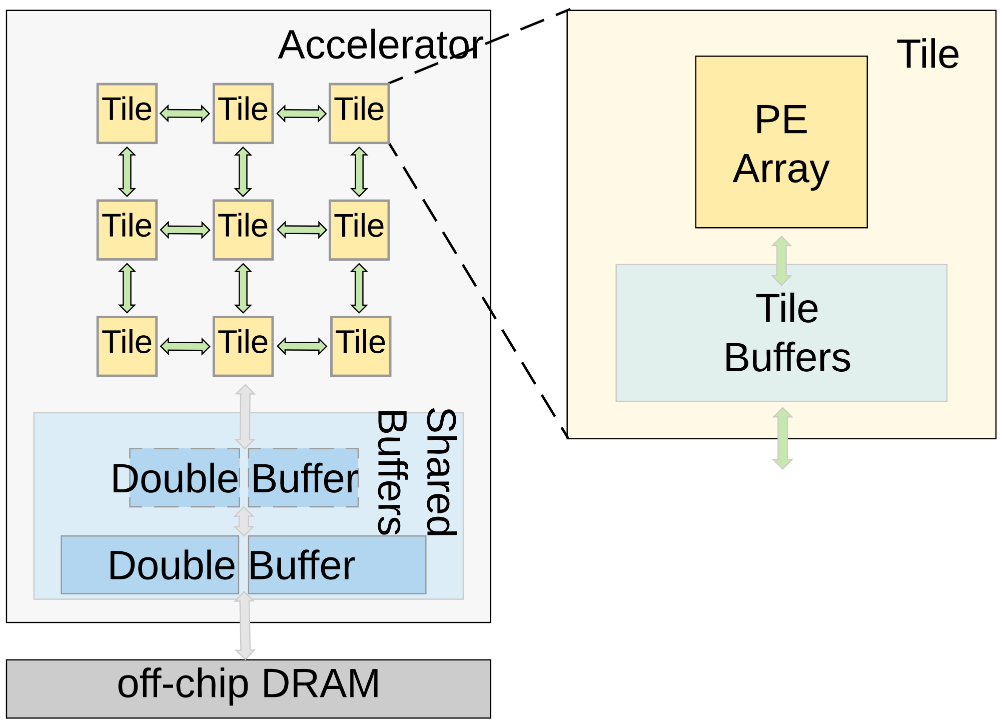
Despite of the wide range of existing accelerators introduced in Section 2.2, we can summarize the architecture adopted by most previous accelerators as a parametrized template, as shown in Figure 3.1. The template is composed of compute tiles and a memory system, while each computation tile is a PE array (Figure 3.1(a)), which can be configured as either a reduction tree (Figure 3.1(c)) or a systolic array (Figure 3.1(b)). Some examples that use the reduction tree templates are Diannao [12], Zhang et al. [99], MAERI [54] and so on. The first two utilizes reduction trees to compute a dot product by breaking down the computation into a set of smaller dot-product that fit on the tile. The last one directly performs 2D convolutions accumulating the output using the reduction tree. The systolic array template can be employed to describe ShiDiannao [26], TPU [48], Eyeriss v2 [14] etc.. A Systolic array approach is more commonly utilized for accelerating DNNs, as it provides good parallelism and efficiently exploits data reuse by serving data from neighbor PEs rather than from more expensive SRAM buffer or DRAM.

The memory system in the template consists of DRAM, a multi-level shared memories, tile buffers and local register files. The multiple levels of shared memories are shared by and serve data to all compute tiles to reduce the expensive access to DRAM. Inside each compute tile are tile buffers that can be accessed by all PEs inside the tile. Each PE also has its own private register file to buffer a small subset data for reuse. Apart from DRAM, the rest memory hierarchy is all double-buffered. While the PEs are operating on the data in one buffer, the data for the next iteration is being loaded to the next buffer, as was introduced in Section 2.4.2.

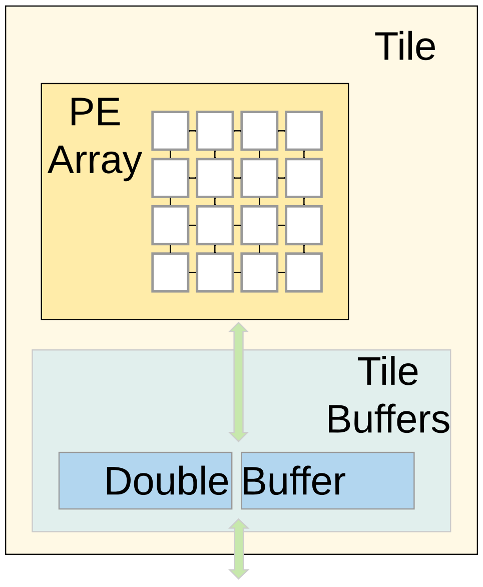
This template is sufficiently comprehensive to describe the architectures of most previous accelerators. Take Eyeriss [13] as one example. The compute core of Eyeriss is one compute tile of a 14×14 2D PE array. Its memory system can be precisely expressed as 512B local register file per PE, 128KB tile buffer and DRAM. No shared buffers are allocated since just one compute tile is instantiated; this situation can be configured by setting the number of shared buffers as zero. A tiled architecture such as Tangram [30], has a 8MB shared memory, which is shared among all 16×16 compute tiles. Each compute tile containing a 8×8 PE array, with a 64 B register file per PE and a 32 kB tile buffer. Other previous proposed accelerators including TPU [48], SCNN [68], Zhang et al. [99], etc. can be concisely described by this template as well, but the details will be omitted here. The next section we will discuss how to efficiently schedule and map DNNs on this architecture template to maximize performance and minimize the energy cost.

3.1.2 DNN Schedules

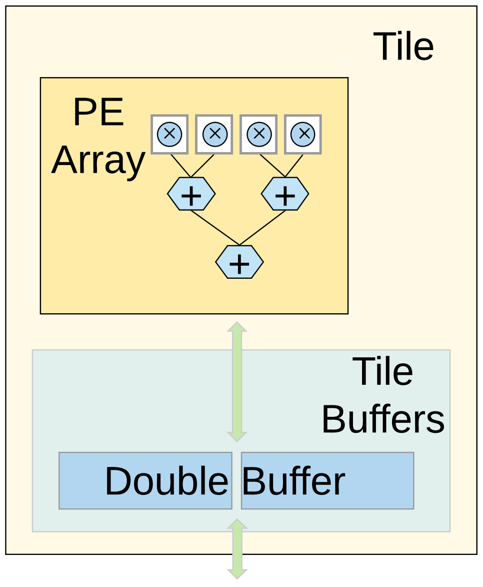
How the DNN is mapped and scheduled on our hardware affects the utilization rate of the computation resource, which directly sets the hardware performance. It also sets our ability to carefully



(a) Architecture overview.



(b) Systolic array.



(c) Reduction tree.

Figure 3.1: DNN accelerator architecture consisting of tiles of PE array and a memory hierarchy.

organize the data movements in the multi-level memory hierarchy to exploit data locality, by moving as many accesses possible to smaller memory. This optimization improves operand bandwidth, and reduces the energy cost of fetching the data.

ASIC accelerators have used a number of efficient scheduling approaches, Diannao adopted a simple schedule which transforms CONV layers into a set of dot products. This ordering allowed accumulating partial sums locally, new inputs and weights to be fetched every iteration, resulting in no data reuse of these variables. Like many other groups, a later design from this same group replaced the expensive data broadcasting from SRAM buffers or DRAM by using a systolic array architecture. ShiDiannao [26] again optimized for output reuse, now by breaking down the ofmap into multiple small tiles and laying out a ofmap tile onto the 2D PE array. Scheduling in this way, each PE accumulates one ofmap pixel, while inputs are transferred and reused among PEs in systolic fashion. Many different schedules can be used on a systolic array. The TPU transformed CONV layers into General Matrix Multiplication (GEMMs) flowing both ifmaps and ofmaps across the array, to optimize the weight reuse inside PEs, as well as the input and output reuse among neighbor PEs. Eyeriss proposed a novel dataflow choice, named as Row Stationary. It breaks CONV layers down into 1D convolution primitives and spatially lays out 2D convolution onto the PE array, by mapping a 1D primitive to each PE. Leveraging Row Stationary's capability of optimizing all types of data reuse, the architecture of Tangram [30] comprises multiple compute tiles, where each implements a Row Stationary dataflow. Among compute tiles, the data communication is optimized by exploring efficient ways to partition data within each layer and across layers.

FPGA accelerators followed similar types of schedules. Zhang et al. [99] also transformed the computation of CONV layers into GEMMs, and instantiated a set of MACs to compute the dot product. They also explored different loop orderings and tilings around the dot product to optimize resource utilization and off-chip bandwidth. Shen et al. [81, 82] further improved the resource utilization by jointly optimize the mapping of multiple layers on multiple instantiated compute tiles. Many other works also optimized the schedules and mappings of DNNs, which will be discussed in Section 3.2

Despite of the large amount of efforts devoted by previous researchers to determine the optimal schedules and mappings, this task has not been fully solved. Three factors make the problem difficult to solve: large schedule space, correlated data reuse to exploit, workload and platform variation.

Large schedule space: The number of schedule choices is enormous. As mentioned in Section 2.1, CONV and FC layers can all be presented as 7 nested loops as presented in Algorithm 1. Scheduling this loop nest can be regarded as performing loop transformations, including loop reordering, splitting, paralleling, etc. These transformation techniques can help to optimize data locality and performance, which will be discussed in more details in Section 3.3. Using loop splitting, we can partition each single loop into a outer and an inner loop. For example, splitting loop `x` results in two new loops `xo`, `xi`. `xo` scan across tiles from 0 to X_o, `xi` iterate inside the tile from 0 to X_i, and $X_o \times X_i = X$. Repeating the splitting process creates more loops which iterate over the same original dimension. Thus the number of choices of loop splitting is numerous, since each loop split has many choices of the tile size (X_i) and each original loop can be split multiple times.

Additionally, when considers on loop reordering, the schedule space becomes even larger. Just considering loop reordering without loop splitting, gives $7! = 5040$, which is the number of all possible loop orders of a 7 nested loops. When loop splitting is considered to better use a two-level or three-level memory hierarchy, each loop maybe split once or twice, the total count can go up to $(7!)^2 \approx 2.5 \times 10^7$ (each loop split once) to $(7!)^3 \approx 1.3 \times 10^{11}$ (each loop split twice). Since loops are split to block for each level of the memory hierarchy, the loop order with a n-level memory hierarchy has $(7!)^n$ options in total. The number of loop orders goes exponentially with levels in the memory hierarchy.

In this large scheduling space, an accelerator designer has additional choices to make : which loops to unroll and execute in parallel on the hardware. For instance, choosing two loops from the seven nested loops to be spatially mapped onto the vertical and horizontal dimensions of 2D array makes $\binom{7}{2} = 21$ options. As a result, considering different choices of paralleling loops further enlarges the schedule space.

In summary, combining all the loop transformation options yields a huge schedule space for a single CONV or FC layer.Considering ways to schedule multiple layers concurrently will further expand this space. Without an understanding of the full design space, it is difficult to figure out the optimal schedules and key scheduling factors. Therefore, it is useful and necessary to have a systematic way to construct the design space for thorough analysis and rapid exploration. This framework is presented in the next section.

Correlated data reuse: The large amount of data and high data reuse requires careful scheduling to maximize data locality. There are trade-offs when optimizing the data reuse for ifmaps, ofmaps, and filters. As presented in Table 3.1, the dimensions of ifmaps are X, Y (image dimensions), C (input channels) and B (images in a batch), to maximize the reuse of ifmaps, ideally we want to hold the ifmaps in the buffer as input stationary and iterate over loops f_x, f_y, k, since each ifmap pixel is reused by all the coefficients in the filter window, and across all filters. However, this schedule results in filter weights streaming through every cycle with no reuse. To fully maximize the reuse of filters weights, instead the filters are kept stationary in the buffer while iterating over loops x, y, b. Due to such trade-offs, optimizing for only one data type reuse is unlikely to be optimal. Instead, finding the optimal schedule demands carefully balancing the reuse among different data types.

Workload and platform variation: The optimal schedules depend on the input layer shapes and the underlying processing hardware. DNN models may have distinct layer configurations, as optimized for a diversity of working scenarios. Even within the same network, layers at different stages are designed and optimized to extract features of varying types and at varying scales, this

Table 3.1: The dimensions of each data block (set D), and loops reuse the data block (set V)

Data	**Dimensions**(D)	**Loops with reuse**(V)
ifmap	X, Y, C, B	f_x, f_y, k
ofmap	X, Y, K, B	f_x, f_y, c
weight	F_X, F_Y, C, K	x, y, b

gives rise to many possible CONV/FC layer shapes. Similarly, targeting at different purpose (energy efficiency, throughput or latency, etc.) and applications (heavy or light workload, image or sequence models), gives rise to significant different platforms and architectures. To achieve high efficiency, the schedule must be optimized individually for distinct layers and platforms. This requires either generating a large number of binary objectives for different schedules, or reschedule the input algorithm rapidly by Just-In-Time (JIT) compilation.

3.2 Previous DNN Scheduling Taxonomy

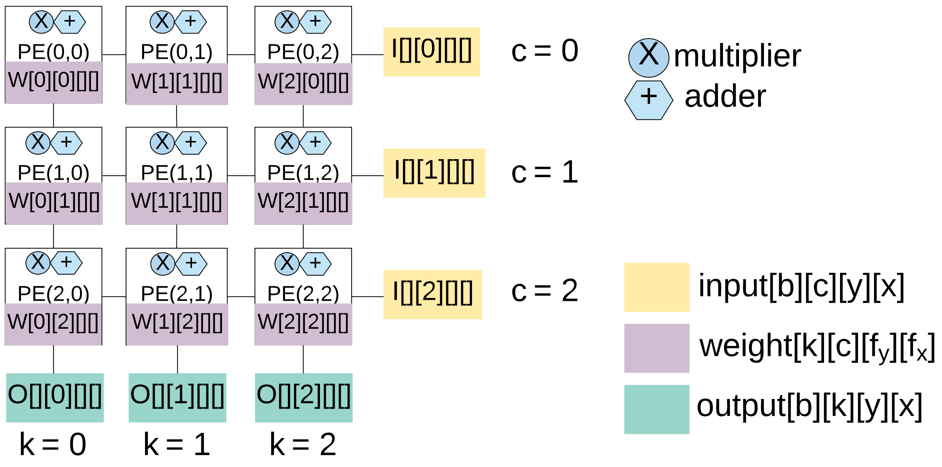
Given the importance of finding optimal schedules and mappings of DNNs, previous researchers attempted to systematically describe the schedule space by proposing different types of taxonomies. These efforts can be broken into three main classes: stationary-based [13], loop nest-based [57] and data-centric [53]. Specifically, stationary-based dataflow taxonomy categorizes each schedule by the type of the data block that is kept stationary inside the local register file (RF) for reuse. Mapping-based taxonomy represents each schedule by directly describing the spatial and temporal mapping of each dimension of data. Loop transformation-based taxonomy simply use the transformed loop arrangement to describe schedule. We will discuss each in the following sections in more details.

3.2.1 Stationary-Based Dataflow

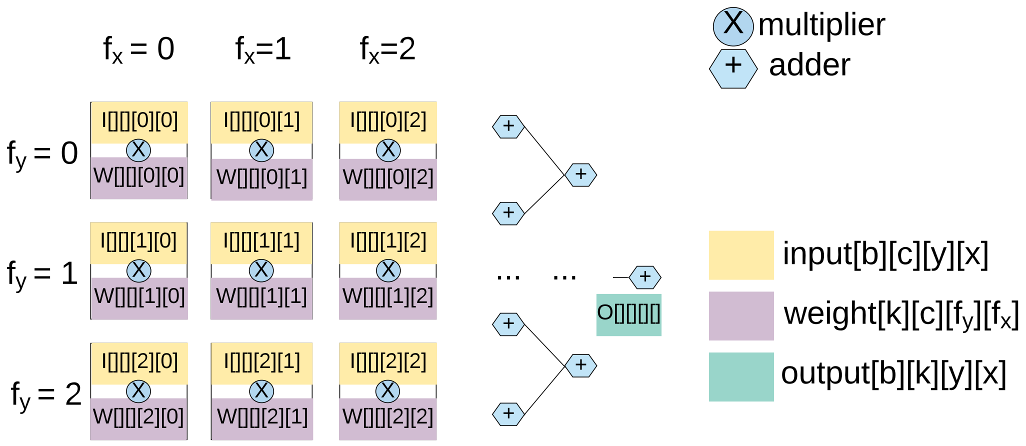
Eyeriss highlighted the importance of on-chip dataflow on energy efficiency and proposes a stationary-based taxonomy to group a variety of prior accelerators based on the type of the data that remains stationary inside the local RF of PEs. As a result, previous accelerators are classified as Weight Stationary (WS), Output-Stationary (OS), Row-Stationary (RS) and Non-Local-Reuse (NLR) [13]. SCNN further adopted and extended this taxonomy, and proposed an Input-Stationary (IS) stationary accelerator [68].

Weight-Stationary (WS) Dataflow: The filter weights are held stationary in each PE's RF. By mapping all operations that use the same weight coefficient onto the same PE for processing sequentially, this datalfow maximizes weight reuse, thereby minimizing the memory energy consumption of fetching weights.

Figure 3.2 presents the data movement of two WS dataflow implementations. For the WS



(a) WS1 dataflow.



(b) WS2 dataflow.

Figure 3.2: Two Weight-Stationary (WS) dataflow implementations. WS1 unrolls input and output channel dimensions (c, k), WS2 unrolls filter width and height dimensions (f_x, f_y). Yellow, purple and green blocks represent the data blocks for input, weight and output respectively. Indexing to the data block is the same as Algorithm 1, for example, $input[b][c][y][x]$ refers to the ifmap pixel at batch b, channel c, row y and column x. Empty index can be either a scalar index or a vector of indices within the valid range.

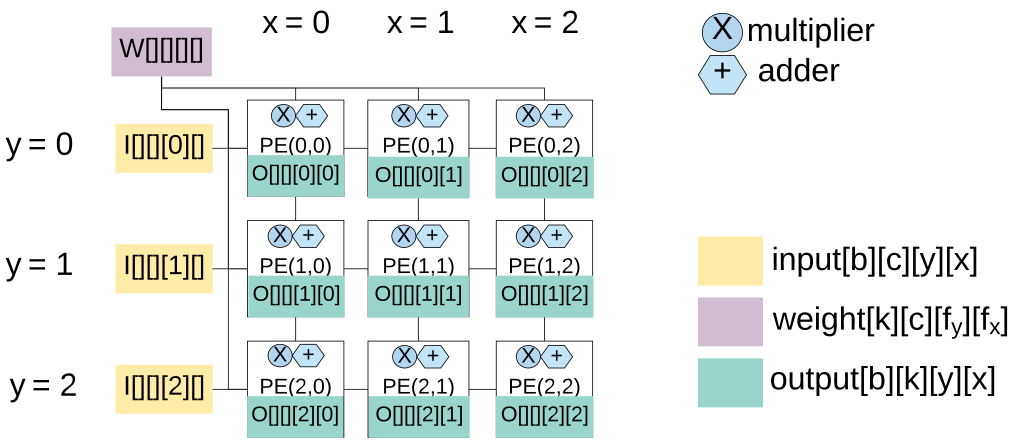
dataflow in Figure 3.2(a), which we name WS1, each weight stays in the RF of each PE, weights from different filters are stored inside PEs at different columns, while weights at different channels are distributed at different rows. Accordingly, the ifmap pixels are broadcasted along the horizontal direction to PEs at the same row, with each channel of the ifmap send to the appropriate row. The computed partial sums (psums) are then spatially accumulated vertically across the PEs at the same column. This specific WS dataflow can be regarded as unrolling and spatially mapping input and output channel dimensions (loop c and k) on the 2D PE array.

Variants of WS1 implementation have been commonly used in previous accelerators, as it is generally flexible to support both CONV and FC layers. Beside, the number of input and output channels of most layers are sufficiently large to maximally utilize computation resource after unrolling. In addition to optimally reuse weight, WS1 dataflow also spatially reuse inputs and outputs. TPU is a systolic version of WS1 dataflow implementation as in Figure 3.2(a), where each PE keeps a unique pair of filter and channel for weight reuse. But instead of broadcasting the ifmap pixels to PEs vertically, each PE relays ifmap pixels to the next PE in a pipelined fashion [48], same for partial sums. Some other systolic array accelerators employing this WS dataflow includes Caffeine [100, 92].

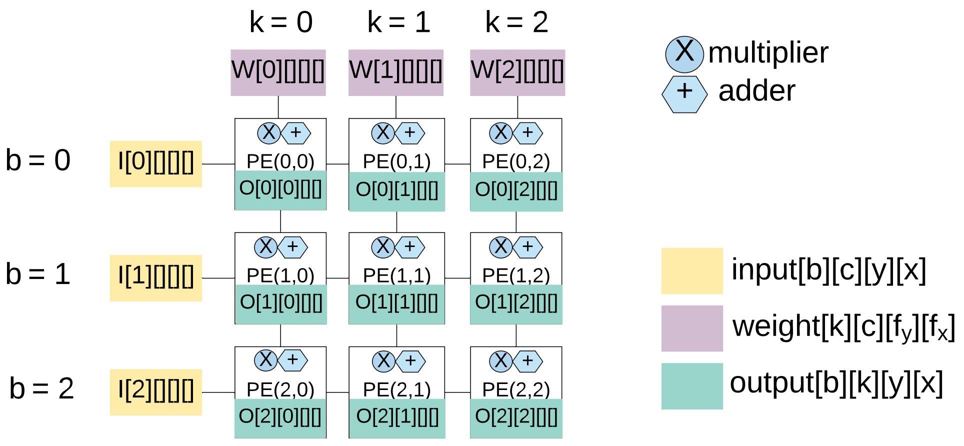
Figure 3.2(b) demonstrates another common WS dataflow implementation, which we will call WS2. $F_X \times F_Y$ weights from the same filter and channel are laid out to a array of $F_X \times F_Y$ PEs, and stay stationary inside PEs' RF. Each iteration, a new set of $F_X \times F_Y$ ifmap pixels are fetched and distributed to be multiplied with the corresponding weights, and accumulates to one output. Essentially it performs a 2D convolution every cycle by unrolling and spatially mapping filter width and height dimensions (loop f_x and f_y).

Qiu et al. [73] and Song et al. [85] are two examples of WS2 dataflow accelerators. For both of them, a window of selected ifmap pixels are read from line buffer and send to the convolver, which are PEs of multipliers followed by an adder tree, to compute the convolution result one data per cycle in a map-reduce fashion. Additionally, Song et al. [85] adopted the WS2 dataflow to accelerate the weight update stages in GANs. To skip the halos (zeros) in the data block, they extended WS to Zero-Free-Weight-Stationary (ZFWS) dataflow that rearranges the data mapping meanwhile keeping weight stationary. Specifically, weights with even indices are grouped together and mapped as neighbors, so do the odd weights. Even though WS2 dataflow can also reuse the partial sums inside the reduction tree, this approach suffers from inflexibility, as layers in DNNs can have different filter window sizes. Mapping a large convolution operation onto a small convolver requires additional control logic and more complex schedules, while mapping a small convolution onto a large convolver simply results in low utilization.

Output-Stationary (OS) Dataflow: The partial sums (psums) are held stationary in each PE's RF to accumulate locally. By mapping all operations that accumulate to the same output onto the same PE for processing sequentially, this dataflow maximizes output reuse, thereby minimizing the memory energy consumption of fetching psums.



(a) OS1 dataflow.



(b) OS2 dataflow.

Figure 3.3: Two Output-Stationary (OS) dataflow implementations. OS1 unrolls feature map width and height dimensions (x, y), OS2 unrolls batch and output channel dimensions (b, k. Yellow, purple and green blocks represent the data blocks for input, weight and output respectively. Indexing to the data block is the same as Algorithm 1, for example, $input[b][c][y][x]$ refers to the ifmap pixel at batch b, channel c, row y and column x. Empty index can be either a scalar index or a vector of indices within the valid range.

Figure 3.3 shows the data movement of two typical OS dataflow implementations. For the first OS dataflow (OS1) shown in Figure 3.3(a), a region of ofmaps are mapped to the PE array and held stationary in the local RF for reuse. Each cycle a new weight coefficient is broadcasted to all PEs, while the required inputs at the corresponding ifmap locations are distributed. This dataflow can be achieved by unrolling and spatially mapping feature map width and height dimensions (loop x and y).

As long as ofmaps are sufficiently large, OS1 can achieve reasonable utilization. Since weights are broadcasted to all PEs, OS1 benefits from weight spatial reuse as well. Besides, as each ifmap pixels contributes to multiple outputs, by carefully arranging the ifmap pixels movement, input locality can also be optimized. For example, ShiDiannao [26] designed an interconnect inside 2D array to shift inputs left-right and bottom-up for spatially reusing the same inputs across multiple outputs. Peemem et al. [70] manually managed input data storage inside multiple memory banks to transpose input data block for better reuse and higher memory bandwidth.

Another common approach to implement OS dataflow (OS2) is illustrated in Figure 3.3(b). It essentially transforms both CONV layers and FC layers into general matrix multiplication (GEMM) $C = A \times B$, then adopts one of the common schedules for GEMM [69]. Specifically, a new column of A is broadcasted horizontally, and a new row of B is broadcasted vertically at each cycle. After K' cycles, where K' corresponds to the inner dimension of the matrix multiplication, each PE accomplishes a dot product between a row of A and a column of B, and accumulates one output of C. This approach is essentially unrolling batch and output channel dimensions (loop b and k).

Since both CONV and FC layers have dimension b and k, OS2 dataflow can be adopted to map both CONV and FC layers. Besides output reuse, OS2 also spatially reuse inputs and weights. For example, Gupta et al. [36] implemented a systolic array employing this dataflow. To better exploit the spatial reuse of inputs and weights, the systolic realization replaces the broadcast communication with access to neighbor PE's RF. Despite the additional reuse, one drawback of OS2 approach is its restriction of the batch size or latency. When low latency is required which takes a small batch size, it potentially lacks sufficient parallelism and leads to low utilization and energy efficiency.

Input-Stationary (IS) Dataflow: Each ifmap pixel is held stationary inside one PE as it is multiplied by all of the filter weights needed to make all of its contributions to a volume of $K \times F_X \times F_Y$ ofmaps. By mapping all operations that utilizes the same input onto the same PE for processing sequentially, this dataflow maximizes input reuse, thereby minimizing the memory energy consumption of fetching ifmaps.

IS dataflow is the opposite of OS dataflow, a region of ifmaps are mapped to PEs and kept stationary, while moving the psum among the PEs. It is adopted by SCNN [68] for accelerating sparse networks, due to their intention to buffer all the compressed fmaps onchip. This objective requires a relatively large input buffer, whose access is expensive and slow. To amortize this energy cost, inputs are registered inside PEs, substituting the more expensive accesses to the larger buffer

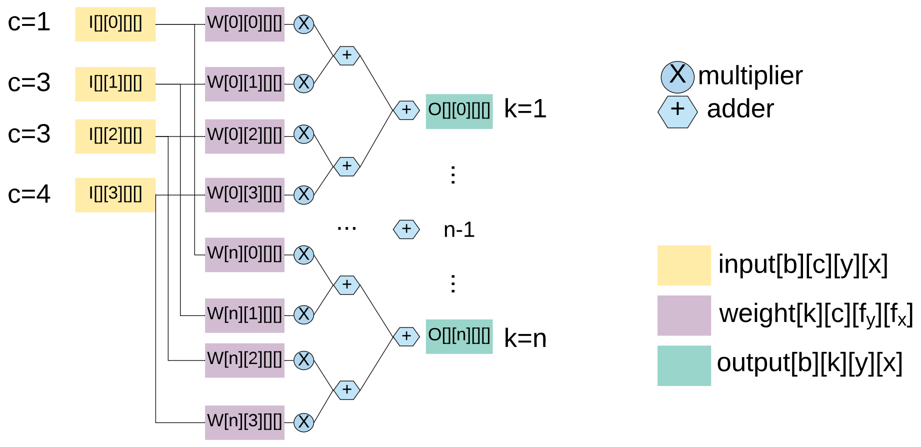


Figure 3.4: The Non-Local-Reuse (NLR) dataflow implementation. It unrolls input and output channel dimensions (c, k). Yellow, purple and green blocks represent the data blocks for input, weight and output respectively. Indexing to the data block is the same as Algorithm 1, for example, $input[b][c][y][x]$ refers to the ifmap pixel at batch b, channel c, row y and column x. Empty index can be either a scalar index or a vector of indices within the valid range.

with the cheaper accesses to registers, while paying a cost to stream the weights to the PE array. Note SCNN utilized a tiled architecture, at the tile level, each tile of ifmaps are kept stationary inside the compute tile as well.

Non-Local-Reuse (NLR) Dataflow: The NLR dataflow does not exploit data reuse at the RF level, but captures the ifmap reuse and accumulates psum using iter-PE communication.

As depicted in Figure 3.4, PE array is divided into groups of PEs, PEs within the same group read ifmap pixels and filter weights from different input channels, and accumulates one psum locally inside the group. Different PE groups share the same sets of ifmap pixels but compute with different filter weights from the same input channel. The architecture is essentially a parallel collection of reduction trees. Within each reduction tree, the input channel dimension (loop c) is unrolled, while output channel dimension (loop k) is spatially mapped across trees.

It maybe surprising to notice that the loop spatial mapping of NLR is the same as the one of WS1, but NLR dataflow doesn't have a RF inside PEs, thus it is less efficient. It can't to exploit input spatial reuse and weight temporal reuse, and only takes advantage of accumulating psums locally using ALU datapaths. Similar to WS1, NLR also preserves good flexibility, variants of the NLR dataflow appear in Diannao [12], Zhang et al. [99], Alwani [8], Chakradhar et al. [10], Shen et al. [81, 82], Suda et al. [87], etc.. In Diannao, registers are allocated following each PE array column to buffer the psums, which further exploits psum temporal reuse. Zhang et al. [99] explored loop tiling choices to maximize the FPGA on-chip resource and off-chip bandwidth utilization, on top of using NLR dataflow. Shen et al. [81, 82] further investigated how to improve the resource utilization and throughput by implementing a tiled architecture, and orchestrate the data mappings among

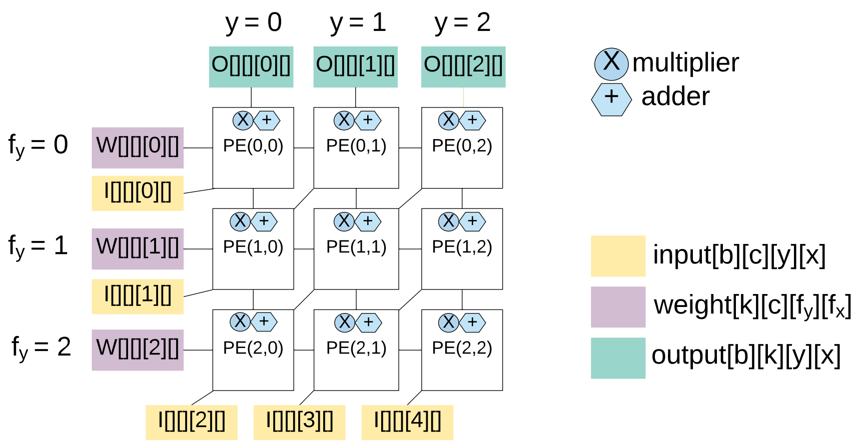


Figure 3.5: The Row Stationary (RS) dataflow implementation. It unrolls filter height and ifmap height dimensions (f_y, y). Yellow, purple and green blocks represent the data blocks for input, weight and output respectively. Indexing to the data block is the same as Algorithm 1, for example, $input[b][c][y][x]$ refers to the ifmap pixel at batch b, channel c, row y and column x. Empty index can be either a scalar index or a vector of indices within the valid range.

tiles. Alwani [8] proposed to fuse and map multiple layers together onchip, thereby eliminating off-chip memory communication for writing back intermediate data between layers.

Row-Statioinary (RS) Dataflow: The RS dataflow divides the MACs into 1D convolution primitives. Each primitive operates on one filter row and one ifmap row to generate one row of psums, thus termed as *row primitive*. By mapping each row primitive onto the same PE for processing in a sliding window order, the computation of each row pair stays stationary in the PE, which exploits the reuse of for all data types combined.

As shown in Figure 3.5, RS dataflow assigns 1D primitives from the same ifmap channel on the PE array to perform a 2D convolution. This mapping capture ifmap, ofmap and filter reuse opportunities at the RF level and across primitives in the NoC. With each PE processing the row primitive in the sliding window order, the filter row and ifmap row are reused temporally inside RF. Besides, each filter row and ifmap row are also spatially reused horizontally and diagonally respectively, while each psum row is vertically accumulated. The mapping can be treated as unrolling filter height and ifmap height dimensions (loop f_y and y).

One simple way to map a RS dataflow on the hardware is to lay out a 2D convolution from the same fmap channel on the PE array. However, frequently there are cases that the 2D convolution is doesn't match the size of the PE array. When the size of the PE array is large, the pattern in Figure 3.5 can be spatially duplicated across the PE array for various 2D convolutions. This technique, named *replication*, increases the utilization of computation resource, but requires more complex interconnects to further exploits data reuse among the PEs. On the other hand, when the size of PE array is small, the original row primitives have to be temporally folded into multiple processing phases. This technique, named as *folding*, serializes the computation and time-multiplexes the hardware for reuse. Both *replication* and *folding* can be adopted for other dataflows as well to improve the utilization or reuse.

Since Eyeriss [13] demonstrated the above benefits of RS dataflow, some of the recent accelerators are also designed using this mapping. Tetris leverages RS dataflow to map NNs onto systems that combines 3D memory with PE arrays. By taking advantage of the high bandwidth and low access energy of 3D memory, it was able to rebalance the NN accelerator design, using more area for PEs and less area for on-chip SRAM buffers [29]. Tangram studied efficient ways to utilize on-chip buffers and optimize intra-layer and inter-layer reuses of DNNs on tiled architecture that supports coarse grain parallelism, with employing RS dataflow inside each tile [30]. Similarly, Hyper also optimized schedules across multiple accelerator to improve both intra-layer and inter-layer parallelism for both inference and training, while using RS dataflow for each accelerator unit [84]. In addition to mapping CNNs, FlexiGAN extended the original RS dataflow to apply to the mappings of GANs as well. The extended RS dataflow can skip halos in transposed convolution stages by data reorganization that groups and maps even rows and odd rows separately [96].

In summary, stationary-based dataflow provides a systematical way to categorize various accelerators based on their characteristics of data mapping. Such approach highlights the importance of data movement through the memory hierarchy, especially local RF inside PEs. By comparing accelerators using different dataflows within this taxonomy, it also demonstrates dataflows that can optimize the reuse for all data types combined such as RS, are generally more energy efficient than other dataflows.

However, this taxonomy has some limitations. First, the schedule choices are blended with the architecture design choices during the classification. More precisely, the categorization of some dataflows are not purely based on the schedule choices, i.e. loop unrolling, but also depends on the existence of RF inside PEs. For example, TPU [48] and Diannao [12] spatially map the same dimensions onto the spatial architecture, but belong to WS and NLR respectively.

Second, the stationary-based taxonomy lacks fine-grain classification. As previously introduced, spatially unrolling different dimensions can generate the same dataflow, even they may have different data movement patterns, flexibility, limitations, etc.. For instance, unrolling c and k loops can generate one WS dataflow, meanwhile unrolling f_x and f_y also keeps weight stationary, this case for other dataflows such as OS.

As a result, it doesn't provide comprehensive and precise representation of the space. It can't describe flows that merge stationary like a hybrid weight and an output stationary pattern is difficult to represent. Finally it omits critical information like replication and splitting.

3.2.2 Loop Nest-Based Schedule

Using a loop nest-based representation is one way to remove the ambiguity of stationary metrics; we used it in the previous section. It has been used to describe some prior accelerators. Loop nest syntax matches the nature of imperative programming language — leveraging loop order and bounds to guide the program execution, and requiring almost no translations from the imperative program to loop nest notation. After applying the loop transformation techniques to the program, including loop blocking (loop tiling, reodering) and unrolling, we can infer potential data reuse opportunities from the resulted loop nests.

For example, Diannao [12] used a 12-nested-loop to express the schedule of CONV layers on their architecture. Zhang et al. [99] adopted a 4-nested loop to represent the external data communication, and another 6-nested loop with explicit loop unrolling to describe the on-chip data mapping and computation.

FlexFlow [57] proposed a taxonomy based on the parallel data dimension. It defined three types of parallelism: feature map parallelism (FP) that unrolls feature map related loops c and/or k, neuron parallelism (NP) that unrolls loop x and/or y, and synapse parallelism (SP) that unrolls loop f_x and/or f_y. Based on the types of parallelism exploited, the schedules are categorized as seven different combinations, including Single Feature map, Single Neuron, Multiple Synapses (SFSNNMS), Multiple Feature map, Single Neuron, Multiple Synapses (MFSNNMS) and so on. This parallelism nest-based taxonomy features moderately good readability, but lacks precise description about how each data dimension is mapped onto the PE array, since unrolling loop c and k both leads to feature map parallelism.

Compared with Stationary-based taxonomy, loop nest-based taxonomy is less well developed. The existing ones only study a subset of the schedule space, or only provide a coarse grain classification. To preform a comprehensive and systematic analysis, we adopt the similar concept of expressing schedules by loop nests, and propose our loop-next based taxonomy to describe the space in Section 3.3.

3.2.3 Data-Centric Dataflow

Masetro [53] proposed a data-centric dataflow representation to explicitly describe data schedule, tiling and mapping on PEs. This is different from loop nest-based dataflow, where loop order, parallelism and tiling implicitly guide data movement and organization across multiple levels of memory hierarchy to influence data reuse. The data-centric dataflow representation is based on three directives: spatial map, temporal map and cluster — the first two encapsulate loop tiling and data mapping over time and space, the last one arranges the granularity of the mapping over space.

Similar to loop nest based notations, the data-centric directives are processed from innermost to outer ones. Instead of using loop unroll primitives, the cluster directive `Cluster(size)` allows the user to describe a mapping over multi-dimensional PE array with clustering of PEs. The base clusters are simply the PEs in the array. `size` determines the number of sub-clusters bundled at the level `cluster` is specified. Using the directive multiple times constructs superclusters, which becomes a unit dimension for outer directives.

The temporal map directive `TemporalMap(size, offset) dim` is used to schedule loop tiling and mapping over time. It maps the same set of data in the dimension to all the sub-clusters, with `size` determining the number of elements mapped in a data dimension `dim` (loop tiling), and `offset` specifying how the mapping move to all of the sub-clusters in the next iteration (loop tile iteration rule over time). For example, `TemporalMap(3, 1) X` maps three elements in X dimension in each iteration, and shifts by one in the next iteration, thus the mapped x over time is (0, 1, 2), (1, 2, 3) and so on.

The spatial map directive `SpatialMap(size, offset) dim` is used to schedule loop tiling and mapping over space. It maps different sets of data in a dimension to all sub-clusters, with `size` determining the number of data mapped in a dimension `dim` (loop tiling), and `offset` specifying how the mapping move to the next cluster (loop tile iteration rule over space). For example, `SpatialMap(3, 1) Y` maps three elements in Y dimension on each cluster (PE in the case), and shifts by one in the next cluster (PE). Specifically, the mapped y index on each PE is: PE0(0, 1, 2), PE1(1, 2, 3) and so on.

Using these three directives, the stationary-based dataflows can be represented in the data-centric representation. For example, the RS dataflow can be represented as below:

$TemporalMap(1,1)k \rightarrow TemporalMap(1,1)c \rightarrow SpatialMap(F_X,1)y \rightarrow Cluster(F_Y) \rightarrow$
$TemporalMap(F_X,1)x \rightarrow TemporalMap(F_Y,F_Y)f_y \rightarrow TemporalMap(F_X,F_X)f_x$

The key advantage of a data-centric dataflow is facilitating the analysis of data reuse, buffer size and access, and thereby easing performance and energy efficiency analysis for better productivity in design space exploration. This is the natural space of the data-centric representation, which explicitly precisely states how each dimension of the data maps and iterates over time and PEs. It also achieves fine-grain classification, for example, distinguishing different WS dataflows by `SpatialMap` directives, and can represent any hybrid dataflows.

However, this representation is difficult for people to create or read and understand. It could be viewed as intermediate representation (IR) of dataflows extracted from loop-nest notations, which we present in next section.

3.3 Design Space Overview

With our formal loop blocking approach to characterize DNN accelerator, we can now cleanly partition its design space along 3 axis: *hardware dataflow* [13, 53], *loop blocking* [95] and hardware *resource allocations*. To obtain more insightful understanding of the design space, we therefore consider a corresponding three-dimensional design space for DNNs, as shown in Figure 3.6. We will later use design space to fairly compare different accelerators.

Dataflow: DNN accelerators often exploit parallelism to improve performance, by using multiple processing elements (PEs) simultaneously. Essentially it executes one or more loops in Algorithm 1 in parallel through *spatial loop unrolling*. The data access and communication patterns across the multiple PEs are determined by the *dataflow* scheme [13], which we characterize by which loops are unrolled as described in the previous section. Typically, the dataflow is carefully orchestrated so that data accesses to more expensive memories, including the storage in other PEs and the large shared buffers, can be minimized. Note, the dataflow terminology we defined here only specifies data spatial mapping across PEs, does not include loop blocking (tiling, reordering) choices, which is separated as another orthogonal dimension for independent analysis.

Resource Allocation: Hardware resource allocations, such as the dimensions of the PE array and the size of each level in the memory hierarchy, are also essential to the performance and efficiency of the accelerator. They determine the computation throughput, the location of the data, and the energy cost and latency for each memory access. For example, since the energy cost and latency of each data access grow with memory size, an efficient design needs to carefully size each memory level to optimally balance buffer being large enough to hold sufficient data for high locality, while being as small as possible to minimize fetch energy.

To concisely and comprehensively describe hardware resource allocation, the representation is mainly based on the following configurations — `MemoryLevels`(integer), `MemorySize`(vector), `BankSize`(vector), `SpatialSize`(vector), `CommunicationMode`(vector). They describe the total number of levels in the memory hierarchy (including DRAM level), the memory size (Bytes) at each level, the bank size (Bytes) of each memory, the number of parallel compute units that can be spatially mapped on, and how data are transferred among the PE arrays, for example, broadcasting or systolic fashion. Except for `MemoryLevel`, which is a scalar value, the rest are all vectors, with elements at position i indicating the memory size, dimensions of parallel units and communication mode at memory level i ($0 \leq i <$ `MemoryLevel`).

Take Eyeriss [13] and Tangram [30] as examples, Eyeriss is composed of a 14×14 broadcast PE array, a 512B RF inside each PE, a 128KB tile buffer and DRAM, its architecture designs can be expressed as Listing 3.1. Since Tangram comprises 16×16 tiles of 8×8 PE arrays, there are 4 memory levels in total — 64B local RF per PE, 32KB buffer per tile, 8MB shared buffer and DRAM, as presented in Listing 3.2. Note, the bank sizes in Listing 3.2 are not the ones used in those designs, dummy numbers are used for this example.

```
MemoryLevels: 3                               # number of memory levels
MemorySize: [512, 131072, Inf]                # size of [RF, tile buffer, DRAM]
BankSize: [512, 8192, Inf]                    # bank size of each memory (optional)
SpatialSize: [[16, 16], 1, 1]                 # 14 x 14 2D array
CommunicationMode: [broadcast, None, None]    # broadcast bus
```

Listing 3.1: Resource allocation description for Eyeriss

```
MemoryLevels: 4                                # number of memory levels
MemorySize: [64, 32768, 8388608, Inf]          # size of [RF, tile buffer, shared buffer,
                                               # DRAM]
BankSize: [512, 4096, 16384, Inf]              # bank size of each memory (optional)
SpatialSize: [[8, 8], [16, 16], 1, 1]          # 8 x 8 PEs, 16 x 16 tiles
CommunicationMode: [systolic, systolic, None]  # systolic array
```

Listing 3.2: Resource allocation description for Tangram

Figure 3.6, used a simplified representation of this data, which gives the total number of MAC units N, and the memory size S_i at each level i, represented as a vector $(N, S_1, S_2, \ldots)$. The full resource representation is used for detailed analysis and space exploration, which will be discussed in Section 5.1.

Loop Blocking: Most DNN layers have high computation intensity, but, like GEMM, the large data sizes require the computation to be properly blocked for efficient execution. What makes the blocking hard is the fact that all the data fetched — input, weight and output are reused multiple times. In a convolutional layer, output pixels at different fmap postions share the same filter; different filters share the same input image; and overlapping windows share ifmap pixels. So any blocking scheme will cause some data to be refetched.

Assuming a multi-level memory hierarchy (e.g., register files, on-chip SRAM, and off-chip DRAM), we want to schedule the computation to maximize the data reuse in the near, smaller memories to lower overall energy cost. However, to reduce the energy cost of the local memory, the local memory size need to remain small. With limited local storage, the data is often kicked out before exhausting reuse, leading to fetching data from higher-cost memory. Thus, it is of great importance to choose proper data block sizes to balance the trade-off between data reuse and energy cost for access data.

Since all the data fetched — inputs, outputs, and weights — can be potentially reused, an optimal schedule must choose the best data reuse opportunities that optimizes the overall locality. We will use the techniques of loop splitting and reordering, which together we refer to as loop blocking [95], to transform the nested loops in Algorithm 1 to optimize the different data access/reuse patterns for each memory level.

Note, in our design space definition, loop blocking is separated from the dataflow terminology, while our dataflow dimension only covers the loop unrolling choices with given blocked loops. This definition decoupling is one of the major differences from other prior works [13, 53, 68]. It allows us to break the cross-product of the two factors, and independently investigate the impact of each one. Leveraging the space shape along each individual dimension, the large space exploration can be better understood and potentially accelerated.

Using these three factors enables us to create a design space that is comprehensive and systematic, enabling us to project existing DNN accelerators in Figure 3.6. We do not explicitly show the loop blocking schemes for each architecture in this figure, since most prior work did not report their loop blocking strategy, or simply exhaustively searched for an ad-hoc optimized scheme. This figure

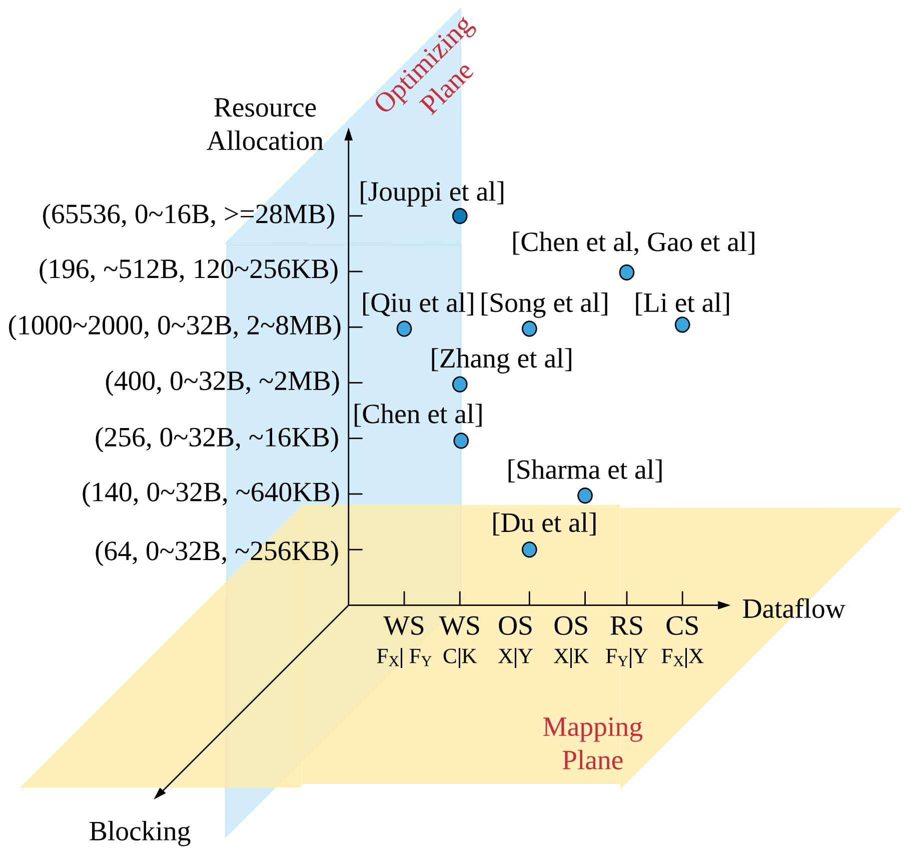


Figure 3.6: 3D design space for DNN accelerators. The positions of labels or vectors on each axis only represent different choices without specific information about ordering or distance.

clearly shows the wide design space current accelerators occupy. The next chapter will show how the modified scheduling primitives in Halide cover not just loop blocking, but also resource allocation and dataflow. This system makes it easy to fairly compare different accelerator design decisions.

3.4 A Formal Loop-based Dataflow Taxonomy

To create a formal dataflow taxonomy based on loops, we first describe the basic blocking notation that we will use, and then show how the dataflow of an accelerator can also be represented by loop operations. The computation being performed by a convolutional layer can be easily expressed as a 7 layer loop nest as shown in Algorithm 1. Since there are no dependencies in this computation except along the reduction dimension, the loops in the algorithm can be done in various orders. We will represent a particular implementation order by creating a vector of string that indicates the loop order from innermost to outer. Thus $[f_x, f_y, x, y, c, k, b]$ presents the computation order shown in Algorithm 1. We create another vector to represent the bound of each of the loop variables so it represents the number of iterations done at each loop level, $[F_X, F_Y, X, Y, C, K, B]$.

Given this initial loop nest, blocking can be thought of as simply splitting a number of loops, and then exchanging the order in which these split loops are executed. In our notation, when the x loop is split, x_0 represents the inner part of the X loop and the loop bound for x_0, X_0, represents the range of the data computed in this loop. x_1 represents the outer loop and X_1 again represents the range of data computed in this loop. In this case, with $X = X_0 \times X_1$, the loop bound remains the same, and the original loop variable x can be represented as $x_1 \times X_1 + x_0$. In other words, every time the loop variable x_1 increments by one, the iteration in x dimension increments by X_0. Multi-level blocking occurs when a single loop is split multiple times, and is easy represented in our notation extending X_1 to X_n, with $X = \prod_{i=0}^{n-1} X_i$.

We use multi-level blocking to create a loop representation that we can map to a 3-level memory hierarchy. As shown in Listing 2, each of the seven loops in Algorithm 1 is split twice to generate three new loops (except filter window dimension f_x and f_y for simplicity). In this way, the large data blocks I, O, W are tiled into smaller blocks to store at the near, smaller memory levels for efficient fetch and reuse. To indicate the different memory levels, we extend our blocking notion by adding a dimension which indicates the memory level. Thus the loop nests in Listing 2 are represented as a 2D vector, the loop order vector is $[[f_x, f_y, x_0, y_0, c_0, k_0, b_0], [x_1, y_1, c_1, k_1, b_1], [x_2, y_2, c_2, k_2, b_2]]$, the blocking size vector is $[[F_X, F_Y, X_0, Y_0, C_0, K_0, B_0], [X_1, Y_1, C_1, K_1, B_1], [X_2, Y_2, C_2, K_2, B_2]]$. The 1D vectors at position i indicate the loop order and bounds at memory level i. If certain loops are not tiled at the level i, then the the value for those loop bounds in blocking size vector at position i will be set as 1. The loop order in Listing 2 can be modified to present other blocking schemes. This corresponds to exchanging certain element positions in loop order vectors.

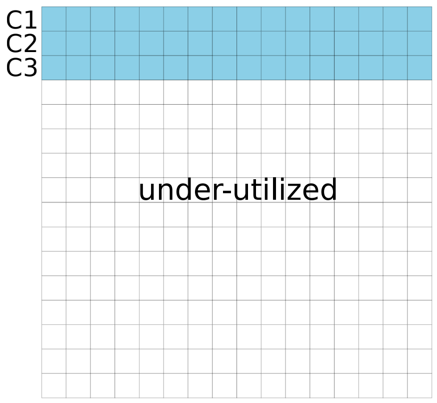
Using the loop order vector and blocking size vector, we can systematically explore the loop

Algorithm 2 CONV layer: Loop blocking for 3-level memory hierarchy.

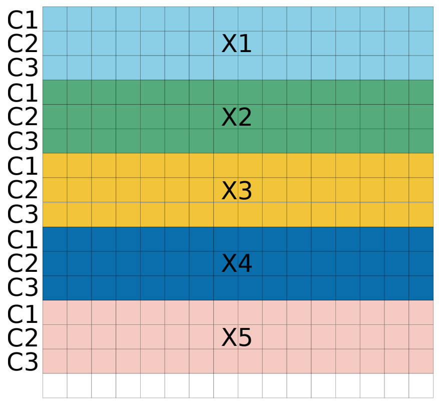
```
==================== {memory level 2} ======================
$\mathbf{I}[B_2 \times B_1 \times B_0][C_2 \times C_1 \times C_0][Y_2 \times Y_1 \times Y_0 + F_Y][X_2 \times X_1 \times X_0 + F_X]$
$\mathbf{O}[B_2 \times B_1 \times B_0][K_2 \times K_1 \times K_0][Y_2 \times Y_1 \times Y_0][X_2 \times X_1 \times X_0]$
$\mathbf{W}[K_2 \times K_1 \times K_0][C_2 \times C_1 \times C_0][F_Y][F_X]$
for $b_2 = 0$ to $B_2 - 1$ do
  for $k_2 = 0$ to $K_2 - 1$ do
    for $c_2 = 0$ to $C_2 - 1$ do
      for $y_2 = 0$ to $Y_2 - 1$ do
        for $x_2 = 0$ to $X_2 - 1$ do
          ============== {memory level 1} ==============
          $\mathbf{I_1}[B_1 \times B_0][C_1 \times C_0][Y_1 \times Y_0 + F_Y][X_1 \times X_0 + F_X]$
          $\mathbf{O_1}[B_1 \times B_0][K_1 \times K_0][Y_1 \times Y_0][X_1 \times X_0]$
          $\mathbf{W_1}[K_1 \times K_0][C_1 \times C_0][F_Y][F_X]$
          for $b_1 = 0$ to $B_1 - 1$ do
            for $k_1 = 0$ to $K_1 - 1$ do
              for $c_1 = 0$ to $C_1 - 1$ do
                for $y_1 = 0$ to $Y_1 - 1$ do
                  for $x_1 = 0$ to $X_1 - 1$ do
                    ======== {memory level 0} ========
                    $\mathbf{I_0}[B_0][C_0][Y_0 + F_Y][X_0 + F_X]$
                    $\mathbf{O_0}[B_0][K_0][Y_0][X_0]$
                    $\mathbf{W_0}[K_0][C_0][F_Y][F_X]$
                    for $b_0 = 0$ to $B_0 - 1$ do
                      for $k_0 = 0$ to $K_0 - 1$ do
                        for $c_0 = 0$ to $C_0 - 1$ do
                          for $y_0 = 0$ to $Y_0 - 1$ do
                            for $x_0 = 0$ to $X_0 - 1$ do
                              for $f_y = 0$ to $F_Y - 1$ do
                                for $f_x = 0$ to $F_X - 1$ do
                                  $\mathbf{O_0}[b_0][k_0][y_0][x_0]$ += $\mathbf{I_0}[b_0][c_0][y_0+f_y][x_0+f_x]$
                                                 $\times \mathbf{W_0}[k_0][c_0][f_y][f_x]$
```

blocking space to find the optimal loop order and tiling size that minimize memory energy by serving most of the data from small memories and minimizing the amount of data that these memories need to fetch from larger, higher energy memories in the memory hierarchy.

Besides loop reordering and tiling, another commonly used loop transformation technique is loop unrolling. Noticing the connection between dataflow and spatial loop unrolling, we represent the dataflow of an accelerator through the mapping of particular loops to the parallel computation structures. In other words, the data communication pattern is determined by *which loops are spatially unrolled* in hardware, and which are not. For example, if the x and y loops are unrolled onto the 2D array, then each PE produces a single output pixel. This output stationary pattern implies that input pixels will be reused across neighbor PEs as they contribute to multiple output pixels in a convolution, and the filter weights shared by all output pixels must be transferred to all



(a) No replication.



(b) With replication.

Figure 3.7: Computation resource utilization can be improved by replication.

PEs. If we instead unroll the F_X and F_Y loops, we obtain a weight stationary pattern, where the weights stay and are reused within the same PEs, but the inputs and outputs are spatially broadcast or accumulated.

To concisely represent dataflows using spatial loop unrolling schemes on 2D PE arrays, we use the syntax $U \mid V$, where U and V denote the loops unrolled across the vertical and horizontal dimensions, respectively. Table 3.2 shows several common dataflows expressed as unrolled loops and the corresponding terminology in prior work. This framework makes it clear that, given L-level (non-trivial) nested loops in the algorithm and d spatial dimensions in the accelerator, there are $\binom{L}{d}$ possible dataflow choices. For a 2D array, the number of dataflow types is $\binom{7}{2} = 21$ for a CONV layer, and $\binom{3}{2} = 3$ for a fully-connected layer.

However, the above considered dataflows, which unroll a single loop at each spatial dimension, can potentially result in under-utilizing computation resources. For instance, as illustrated in Figure 3.7(a), when unrolling a loop C with size of 3 on the vertical dimension of a 16×16 PE array, only 3 of the 16 rows of PEs are utilized, leaving the remaining PEs idle. To overcome this issue, in addition to unrolling loop C, another loop such as X is also unrolled by a factor of 5, as shown in Figure 3.7(b), improving the utilization ratio from 3/16 to 15/16. Therefore, it is of great importance to support unrolling multiple loops onto one spatial dimension. As mentioned earlier, this improvement is called *replication* [13, 14], which processes multiple small loops in parallel to increase resource utilization. Our loop-based taxonomy can also nicely and consistently express it using $U \mid VW$ or $UW \mid V$, depending on the replicated dimension. When replication is supported, the number of dataflow choices increases to $\binom{7}{x}$ for a CONV layer mapping to a 2D array, where

Table 3.2: Common dataflows from [13] expressed using spatially unrolled loops.

Dataflow	**Representation**
Output stationary	$x \,|\, y$
Weight stationary	$f_x \,|\, f_y$
Row stationary	$f_y \,|\, y$
Weight stationary	$c \,|\, k$

Table 3.3: Categorization of prior DNN accelerators using our dataflow taxonomy. Dataflows are represented as $[U \,|\, V, W \,|\, Z, ...]$, where element at position i indicates the unrolled loops at memory level i.

Dataflows	Previous design
$[x \mid y]$	[26, 85, 70]
$[x \mid y, x \mid y]$	[68]
$[x \mid k]$	[80]
$[b \mid k]$	[36]
$[f_x \mid f_y]$	[73, 85]
$[c \mid k]$	[12, 48, 99, 8, 81, 82, 87, 92, 10]
$[f_y \mid y]$	[13, 15, 29, 30]
$[f_x \mid x]$	[56]

$x >= 2$, thus dataflow design space becomes even larger.

Note that with replication (i.e., mapping multiple loops onto the same physical dimension), the data communication pattern is no longer uniform: intra-loop data can be communicated among nearest-neighbor PEs, inter-loop data have to be sent to multiple hops away with higher communication cost. Syntactically, we represent this by ordering the loops mapped to the same dimension, where the PEs generated by unrolled loops to the left have shorter communication distances than the loops on the right. Figure 3.8 shows an example of unrolling both C and K loops onto a 1D array. The eight PEs have been divided into two groups, each working on a output channel K_i. Within each group, different PEs process different input channels C_i. The outputs are only communicated among the nearest PEs, while the inputs have to transfer from one group to the other with a cost four times of nearest neighbor communication.

Replication can improve the resource utilization. However, layers with different configurations (for example, filter window sizes, fmap sizes) may require different optimal replication. Thus flexible or re-configurable interconnect is necessary to support mapping multiple layers on the same compute array. Eyeriss V2 [14] proposed a hierarchical mesh interconnect that can operate at different modes to support a wide range of cases, including high bandwidth, high reuse and so on. MAERI [54] designed a programmable interconnect that can support various DNN layer partitions and mappings by properly configuring the switches in the tree structure. On the other side, the reconfigurability

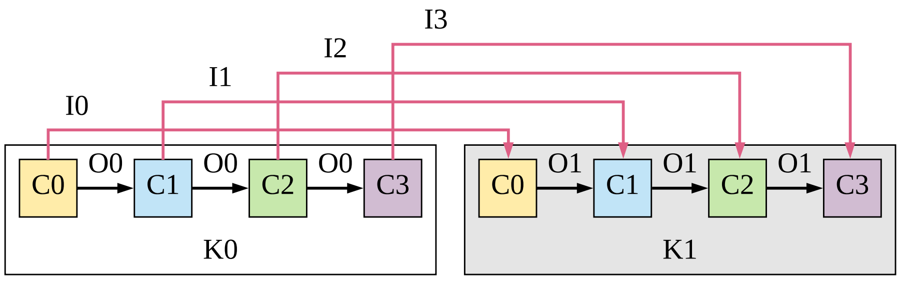


Figure 3.8: Unroll two loops C and K onto a 1D array with dataflow CK. Outputs are communicated between adjacent PEs, while inputs are communicated across groups.

and flexibility of FPGA and CGRA make them good candidate to map the widely varying layers efficiently.

Next we introduce how our taxonomy is capable of expressing dataflows with multi-level parallelism. With tiled architecture such as Tangram [30] SCNN [68], the dataflow choices need to be made both inside the tile and across compute tiles. In this case, the dataflow choice is formed by hierarchically unrolling loops at corresponding memory levels. We represent the hierarchical dataflow as $[U \mid V, W \mid Z, ...]$, where element at position i indicates the unrolled loops at memory level i. For example, given a tiled architecture with 3-level memory hierarchy as shown in Algorithm 2, assuming memory level 0 (RF) is local to each PE, and memory level 1 (buffer) is shared by PEs within one tile, but private to each tile, then spatial loop unrolling choices are explored at both memory level 0 and 1. If c_0 and k_0 loops are unrolled within each compute tile to keep weight stationary, while fmap are split to fmap tiles to map onto different compute tiles by unrolling x_1 and y_1 loops, the dataflow choice is expressed as $[c \mid k, x \mid y]$.

Advantages: The loop-based approach builds upon the ideas of [13] stationary characteristics, and provides a more precise definition of each dataflow while expanding the range of flows that can be described. This benefits can be illustrated from Table 3.3, which categorizes existing DNN accelerators based on this taxonomy. For example, Diannao [12], TPU [48] and Zhang et al. [99] can all be represented as $c \mid k$. Eyeriss, Tetris and other RS accelerators are expressed as $f_y \mid y$. ShiDiannao [26] and Song et al. [85] are categorized as $x \mid y$. For architecture with both fine-grain and coarse-grain parallelism like SCNN [68], the hierarchical dataflow choice is presented as $[x \mid y,\ x \mid y]$. From Table 3.3 demonstrates this taxonomy can precisely express a wide range of prior accelerators. Due to the nature of loop-based taxonomy, the representation can be easily scaled to express spatial mappings on high-dimensional array, or compute substrate with multi-level parallelism.

This taxonomy also provides concise and fine-grain categorization, beyond covering the wide range of existing accelerators. Table 3.2 shows several common dataflows in prior work expressed using our taxonomy. As an example, $C \mid K$ is a widely adopted dataflow (Figure 3.6, and [12, 48, 99, 8, 81, 82, 87, 92, 10]) due to its flexibility to also map matrix multiplications in MLPs and LSTMs. Even though $C \mid K$ also keeps the weight stationary in PEs, its data reuse pattern is quite different from the weight-stationary dataflow $F_X \mid F_Y$ introduced by [13]. Furthermore, more complicated dataflows, such as a hybrid weight and output stationary pattern, are easy to represent in our taxonomy, e.g., $C \mid KX$, demonstrating its completeness.

More importantly, the taxonomy decouples schedule from the underlying hardware configuration. For example, the loop unrolling of NLR [12] and WS1 [48] dataflow are the same, the only distinction is the local RF allocation. Noticing this, our taxonomy categorizes both NLR and WS1 as $c \mid k$, decoupling the hardware from how it is scheduled and resources allocated. The decouplings between schedule and architecture, and between loop blocking and unrolling, makes it possible to fairly compare various designs, and obtain deeper and thorough understanding of the design space.

Through the loop-based dataflow taxonomy, dataflows and loop blocking schemes can now both be expressed as transformations of the seven nested loops in Algorithm 1. There are many approaches for finding loop transformations that optimize some cost functions, and can be used to help find optimal hardware. These include general approaches like Polyhedral analysis [62, 107] and studies specific to DNNs [95]. Taking advantage of the loop-based taxonomy and vector-based representation, we develop an optimizer to systematically search over the huge space for optimal schedule with given or optimized memory systems to minimize energy efficiency while maintaining high throughput. More details about the optimizer will be provided in Section 5.1.

Chapter 4

DSL System Design for DNN

Despite of the enormous design choices for DNN accelerators, we have constructed an abstract space to cover and express them, as introduced in Chapter 3. All the dimensions composing the design space — resource allocation, dataflow and loop blocking choices — can be concisely and comprehensively expressed as loop transformations by the loop-based dataflow taxonomy. This loop-based notation nicely aligns with the Halide scheduling language, which is also based on the loop transformation concept to specify the ordering of operations and data references. As previous work has successfully mapped the Halide scheduling primitives to hardware implementations [72], it provides a way to project an abstract design space to actual hardware implementations. Hence, we also extend Halide to generate DNN hardware accelerators in addition to generating code for CPUs and GPUs. This tool raises the level of abstraction above current high-level synthesis (HLS), providing a DSL system that can translate the high-level algorithmic descriptions to efficient hardware designs with the capability of exploring various design choices.

To fully support the design space described in Chapter 3, we made an extension to Halide, by adding the systolic scheduling primitive, to specify a systolic array architecture. By leveraging Halide compact scheduling language with the small extension, we can describe all hardware architectures and software scheduling choices in the design space, and fairly compare various design choices. Section 4.1 starts with introducing how the extended Halide scheduling primitives correspond to each of the three design space dimensions. To translate the scheduled Halide program to to hardware implementations, we extended the existing Halide compilation system. In Section 4.2, we present the extended Halide IR that can be used by the Halide compiler to describe and optimize a DNN accelerator. Next, we describe the compiler implementation that can generate HLS code from the transformed IR by constructing accelerators from the architectural templates.

The common building blocks in our architecture templates include double buffers and PE arrays, the latter one can be configured as reduction tree or systolic array. To simply the code generation in the Halide compiler, we create support libraries implementing double buffers and systolic arrays,

Table 4.1: Halide scheduling primitives that control each dimension of the 3D design space.

Dimensions	**Scheduling primitives**
Loop blocking	`split`, `reorder`
Resource allocation	`in`, `compute_at`
Dataflow	`unroll`, `systolic`
Overall scope	`accelerate`

which can be used to create hardware instances. Based on the specification of the two modules, Section 4.4 first describes their micro-architecture designs and defines the interface abstractions, and then presents the HLS template implementations that can be configured by scheduled programs.

4.1 Halide Schedules for DNN Accelerators

Halide scheduling primitives can be used to control each of the three design space dimensions as summarized in Table 4.1. *Loop blocking* dimension determines the data reuse opportunities by performing loop splitting and reordering, which can be accomplished by `split` and `reorder` schedule primitives. *Resource allocation* affects the amount of data each buffer can hold and the access cost in the memory hierarchy. Using primitive `in` combined with `compute_at` enables programmers to instantiate buffers and manage the granularity of storage allocation. Based on the same underlying data buffering principles, these primitives can be leveraged to specify the organizations of the memory systems for hardware generation. Lastly, in our loop-based dataflow taxonomy, dataflow is represented by the spatial unrolled loops, thus schedule primitive `unroll` can be leveraged to express *dataflow* dimension, in addition to control the computation throughput. We also add a new primitive `systolic` as an extension for generating systolic array rather than a reduction tree architecture.

Listing 4.1 shows an example Halide code for a CONV layer, which has a 5×5 convolution window and 3 input channels. We use this example schedule to demonstrates how the primitives in Table 4.1 can be adopted to specify the design choices (loop blocking, dataflow and resource allocation) in the space for the written algorithm.

```
// To perform a 5 x 5 convolution with 3 channels
// RDom(xMin, xExt, yMin, yExt, kMin, kExt)

Var xo, yo, xi, yi;
RDom r(-2, 5, -2, 5, 0, 3);

// Algorithm
output(x, y, k) = 0;
output(x, y, k) += input(x + r.x, y + r.y, r.z)
                 * w(r.x + 2, r.y + 2, r.z, k);

// Schedule
output.update()
      .split(x, xo, xi, 8)
      .split(y, yo, yi, 8)
      .reorder(xi, yi, xo, yo)

input.in().compute_at(output, xo);
w.in().compute_at(output, xo);

output.accelerate({input, w});
output.update().unroll(xi, 4);
output.update().systolic({xi});
```

Listing 4.1: An example Halide algorithm and schedule code for a CONV layer.

The algorithm and schedule provide a user-facing language to construct hardware. Figure 4.1 pictorially shows this example schedule in detail; from left to right, we iteratively apply three sets of scheduling primitives to achieve the final accelerator structure on the far right of Figure 4.1. Listing 4.2 and Listing 4.3 presents the intermediate representation (IR) generated by Halide, corresponding to the second and third phase in Figure 4.1 respectively. Next, we explain in depth the usage of each of these scheduling primitives.

```
//To generate output of size 16 x 16 x 64
for (k, 0, 64)
  for (yo, 0, 2)
    for (xo, 0, 2)
      for (yi, 0, 8)
        for (xi, 0, 8)
          for (r.z, 0, 3)
            for (r.y, -2, 5)
              for (r.x, -2, 5)
                output(xi, yi, 0) += input(xi + r.x, yi + r.y, r.z)
                                   * w(r.x + 2, r.y + 2, r.z, 0)
```

Listing 4.2: Intermediate representation generated by Halide after using `split` and `reorder`, corresponding to the second phase in Figure 4.1.

```
 1 //To generate output of size 16 x 16 x 64
 2 for (k, 0, 64)
 3   for (yo, 0, 2)
 4     for (xo, 0, 2)
 5
 6       // Allocate local buffer for input.
 7       alloc ibuf[8 + 5 - 2, 8 + 5 - 2, 3]
 8       // Copy input to buffer.
 9       ibuf[...] = input[...]
10
11       // Allocate local buffer for w.
12       alloc wbuf[5, 5, 3, 1]
13       // Copy w to buffer.
14       wbuf[...] = w[...]
15
16       for (yi, 0, 8)
17         for (xi, 0, 8)
18
19           for (r.z, 0, 3)
20             for (r.y, -2, 5)
21               for (r.x, -2, 5)
22                 output(xi, yi, 0) += ibuf(xi + r.x, yi + r.y, r.z)
23                                   * wbuf(r.x + 2, r.y + 2, r.z, 0)
```

Listing 4.3: Intermediate representation generated by Halide after using `in` and `compute_at` combined with `split` and `reorder`, corresponding to the third phase in Figure 4.1.

Loop blocking: The existing Halide scheduling primitives are primarily designed for loop transformation optimizations on general purpose processors, but the syntax and semantics also support loop blocking on accelerators thanks to the same underlying principles. Lines 13–16 of Listing 4.1 use `split` to break the `x` and `y` loops into two levels, where the inner loops (`xi` and `yi`) have 8 iterations. `split` can also be applied repeatedly to create more levels. `reorder` performs loop interchange, setting the order of computation and data access. These two primitives can realize different *loop blocking* schemes, splitting the data into multiple smaller subtiles (4 tiles of 8×8 in this example) that are processed in a certain order (`x` then `y`). The first step in Figure 4.1 visually shows the four tiles that are created due to the new loops (lines 3–4 and 16–17 in Listing 4.3).

Resource allocation of memory hierarchy: After splitting the data, we need to allocate SRAM buffer resources so that each data subtile can be cached on-chip while being processed to reduce the access cost. The existing primitive `in` is designed to create data copies that can be buffered at other locations in the hierarchy. `compute_at` specifies the granularity of the realization of the caller function. When used together they explicitly control the copy, assignment and movement of data blocks. Combined with loop blocking techniques these commands enable one to create data tiles that can fit in each level in the hierarchy. For CPUs and GPUs these commands reduce cache misses,

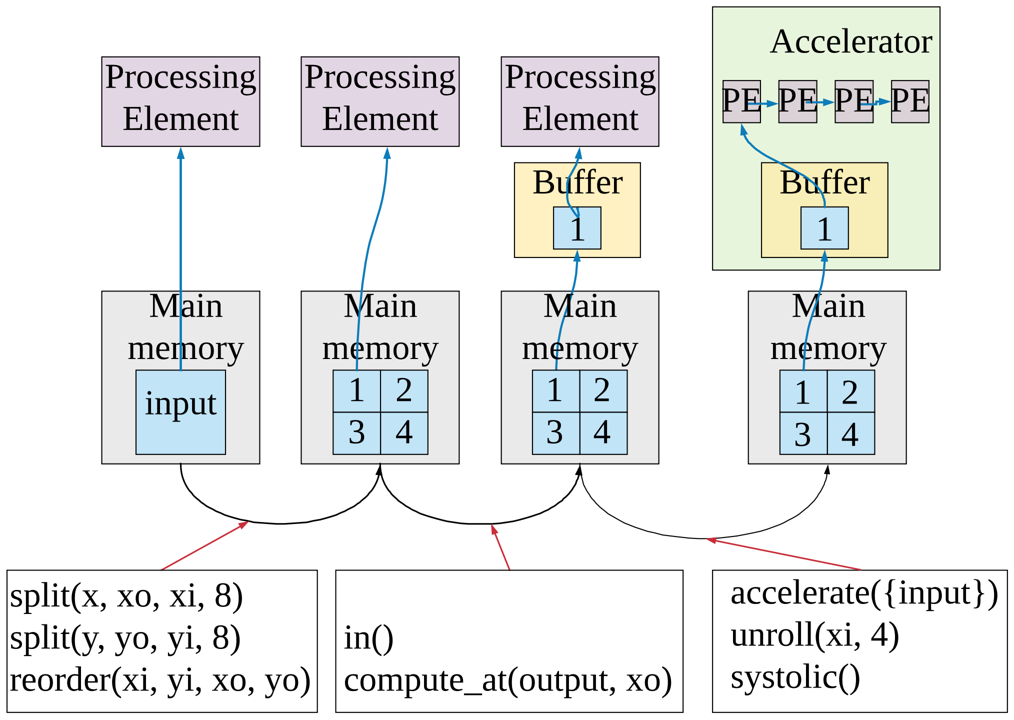
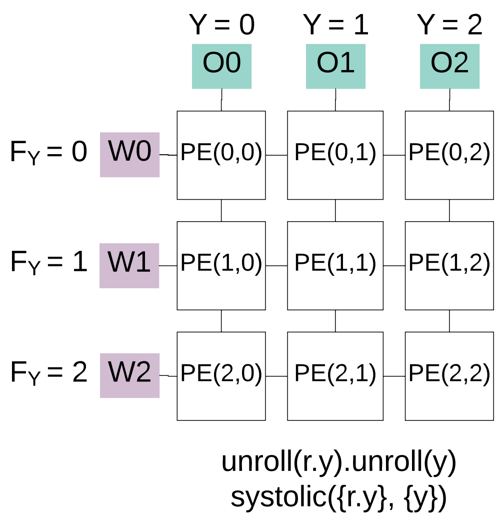


Figure 4.1: The initial design fetched the data as one large block from memory. After the split and reorder, the data is broken into 4 smaller tiles. Next a local buffer for one tile is allocated, and finally a 4 PE systolic array is implemented to process the data.

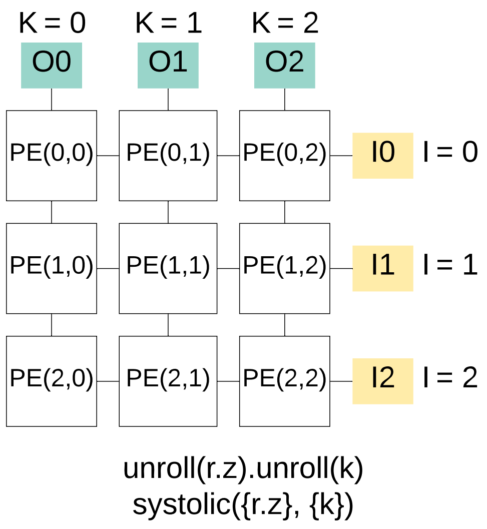
while for generating specialized hardware, these primitives are used to specify the organization of the memory system. The existing primitives `in` and `compute_at` (lines 18–19 of Listing 4.1) together introduce additional memory levels, and specify at which loop iteration to fetch which subtiles into the buffers (Figure 4.1). The compiler combines this information with loop sizes, and instantiates the correct number of memory levels with appropriate size and data layout for each buffer. As shown on lines 6–14 in Listing 4.3, by calling `in` and `compute_at` together for both `input` and `w`, two local buffers are allocated within loop `xo` with the right size to store the input and weight data respectively. By repeatedly calling `in` with `compute_at`, additional memories in the hierarchy can be created, such as a register file inside the inner loops to further optimize memory energy consumption. This allows us to explore different *resource allocation* choices. For each memory level, we use a *double buffer* design (Figure 3.1), which enables overlapping computation and data fetch, by supplying data to PEs out of one buffer while performing loads/stores on the other.

Dataflow and PE array micro-architecture: While the previous two dimensions can be covered using existing Halide primitives, expressing *dataflow* requires extensions to support the complex on-chip data propagation patterns uniquely appearing in accelerators. First, like Pu et al. [72], we overload the existing `unroll` primitive to specify spatial loop unrolling onto the PE array. For the CPU/GPU target, `unroll` is used to eliminate the control overhead of the short loops, and enable the optimizations of data sharing across loop iterations. Pu et al. [72] overloaded this primitive to duplicate hardware resources for supporting variable rate pipelines and space-time trade-off explorations. Noticing the connections between *dataflow* and the effects of `unroll` after overloading, we use it to specify dataflow choices. As discussed in Section 3.4, given a 1D dataflow U or 2D dataflow $U \,|\, V$, we spatially unroll the loops U and V on each physical dimension, respectively. For example, in Figure 4.1, the loop `xi` is unrolled to execute in parallel on 4 PEs. To generate multiple compute tiles, we can further spatially unroll outer loops W and Z to implement dataflow $U \mid V, W \mid Z$. For the example in Figure 4.1, if additionally unroll `k`, multiple compute tiles are created to produce different output channels simultaneously, with each convolving with a different filter by the 4 PEs inside.

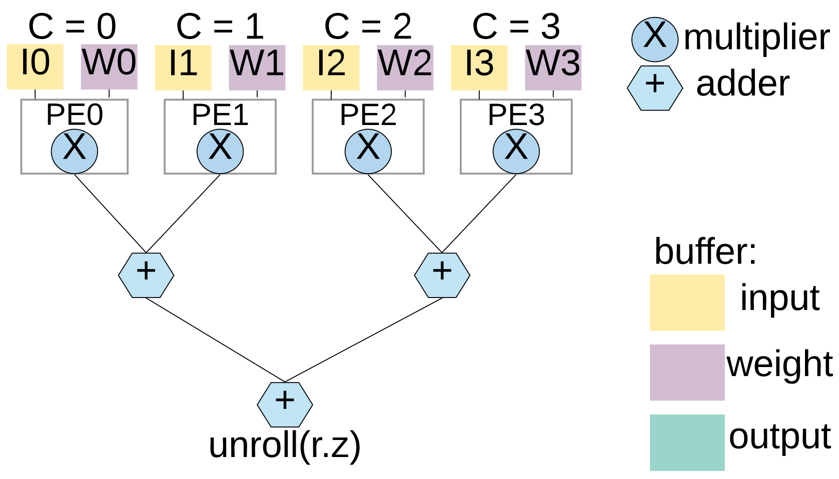
Second, we support various types of PE array micro-architectures. We introduce a new primitive, `systolic`, which realizes a systolic PE array as shown in Figure 3.1(b), and allows direct inter-PE data communication without always fetching data from higher-level data buffers [13, 48]. Note, the interpolation of `systolic` is different from the one of `unroll`. `systolic` specifies the type of the micro-architecture, i.e. systolic array or reduction tree, and the physical mapping for the unrolled dimensions, i.e. which loop is mapped vertically, but it is not responsible for spatially unrolling the loops. This also implies `systolic` primitive should only be applied when `unroll` is used, since `systolic` controls the communication mechanism among PEs, it is only valid when multiple PEs exist. Besides, `systolic` is constrained to be applied when there exist data reuse along the unrolled dimensions for the given computation.



(a) A systolic dataflow $F_Y \mid Y$, generated by unrolling the output fmap height Y and filter weight height F_Y.



(b) A systolic dataflow $C \mid K$, generated by unrolling the input and output fmap dimensions C and K.



(c) A reduction tree of dataflow C, generated by unrolling the input fmap dimension C.

Figure 4.2: Different PE array micro-architectures generated from Halide scheduling primitives.

Next, we describe the effect of `systolic` by showing the accelerator instances corresponding to the presence and absence of this primitive respectively. Using `systolic({U}, {V})` with unrolling U and V loops specify the dataflow $U \mid V$, which maps each loop on a physical dimension of a 2D PE array. To improve the resource utilization using replication, the replicated loops can be added to the list. For instance, dataflow $U \mid VW$ can be specified using `systolic({U}, {V, W})`. If only one argument exist, such as `systolic({U})`, it generates a 1D PE array and maps the algorithm using dataflow U.

By combining `systolic` with different `unroll` primitives, we can realize different dataflows on the PE array. Figure 4.2(a) maps the $F_Y|Y$ dataflow used in Eyeriss [13], by unrolling the F_Y and Y loops. It transfers multiple rows of filter weights horizontally, and accumulates multiple rows of output fmaps vertically. Alternatively, Figure 4.2(b) performs matrix multiplications using dataflow $C|K$, which is used by a large group of designs including Google's TPU [48].

Without applying `systolic`, the PEs are by default organized into reduction tree structures [12], as shown in Figure 3.1(c). Figure 4.2(c) provides one example dataflow on a 1D reduction tree, which unrolls the loop C to multiply input pixels from different input fmaps with their corresponding weights, and accumulates the products into a single output pixel in an output fmap.

Using the two micro-architectures in Figure 3.1 as building blocks, we can generate a variety of accelerator designs by composition. They can be described by applying `unroll` at different loop levels, and calling `systolic` with passing in corresponding unrolled loops as arguments. For instance, we can have a reduction tree of PEs acting as a node in a systolic array, or multi-level reduction trees, effectively supporting a wide range of designs including ARM ML processor [1] and NVDLA [3]. Other PE array styles can also be implemented similarly by applying additional primitives.

Accelerator scope: Finally, we introduce an additional primitive, `accelerate`, which defines the scope of the hardware accelerator and the interface to the rest of the system, in a similar manner to Pu et al. [72]. `f.accelerate(inputs)` specifies the accelerator scope as a part of DAGs that takes a list of input functions (`inputs`) to produce function `f`.

4.2 Extended Halide IR

Given this concise and concrete expression for DNN accelerators using Halide schedules, we extended the Halide system to generate hardware from these descriptions. As a result, different DNN accelerator designs and mapping schemes can be realized by simply changing the Halide schedule associated with the same Halide algorithm. We can also use it to easily recreate previously proposed designs for a fair comparison (see Section 5.3).

The Halide compiler translates an algorithm with given schedule to an imperative intermediate representation (IR) before code generation. An IR program represents the syntax tree, whose nodes include memory allocations, algorithm expressions, control flows and so on. It is useful for compiler to analyze the characteristics of the program, and apply optimizations, including common optimizations (such as constant propagation) and domain specific optimizations (sliding window and storage folding for creating software line buffer).

However, the Halide IR can not be directly used to generate hardware as the memory models are different: software running on general purpose processors can leverage its cache to manage the underlying data movement; hardware generation requires designing the memory structures and manually managing the data movement in the hierarchy to optimally exploit data locality. To address the issues of using Halide IR, Pu et al. [72] present a new IR, the *dataflow IR*, by extending the existing Halide IR. It is used by the compiler for describing and optimizing the hardware accelerator generated from the line buffer architecture template.

To allow generating DNN accelerators, we further extend the *dataflow IR* to support expressing and optimizing an accelerator constructed from a double buffer architecture template. To avoid confusion with the DNN dataflow described in Chapter 3, we call this new IR the *extended IR* instead of the *dataflow IR*. Similar to the *dataflow IR*, the extended IR also uses stream interfaces for data communication between pipeline stages, but we further extend it by adding two new intrinsic functions for the double buffer and the systolic array modules as required by our DNN architecture template.

When creating a new hardware memory structure, the memory interface is often required to be sufficiently wide to provide high bandwidth matching the computation throughput. For instance, to compute a 3×3 2D convolution at one pixel/cycle rate, the required interface width should be nine pixels. Since in the existing Halide IR, load and store operations are defined at a pixel granularity, Pu et al. [72] introduces a *stencil stream*, with two data types, *stencil* and *stream*, to express the desired data width of the stream interface in the IR. We leverage the idea and the design of *stencil stream* for developing the double buffer pipeline for DNN accelerators.

A stencil denotes a multi-dimensional array of elements, with supporting random access. It is implemented as a fully-partitioned array, and synthesized to a set of registers, each storing a pixel. It allows reading or writing multiple elements simultaneously within one clock cycle, thereby providing high memory bandwidth. A stream is a streaming interface for transferring data in the FIFO order. A stencil stream can pass a complete stencil each cycle, the implemented FIFOs are as wide as the aggregated stencil width. We reuse the same IR as Pu et al [72] for stream operations.

- `def_stream(str, t, extent)` declares a stream `str` of data type `t`. `extent` is a list of integers expressing the extents of the stencil dimensions. If it is just a scalar stream, the extend is $[1, 1]$.
- `pop(var, str)` reads stencil `var` from stream `str`.
- `push(var, str)` writes stencil `var` to stream `str`.

In the extended IR, we also add two new intrinsic functions, *double buffer* and *systolic array*, to facilitate code generation. In the IR, the input and output data of these two blocks are stream data type, and other parameters to configure them are encoded as constant arguments for the functions. Adding these intrinsic function abstractions reduces the complexity of code generation, since these functions can be implemented in a support library: the code generation only needs to create the instances constructed from the library.

- `doublebuffer(write_stream, read_stream, write_addr_stream, read_addr_stream, write_bound, read_bound, buffer_size)` specifies a double buffer with a constant size `buffer_size` for each of the two banks. Data from the current data block is send from one bank to computation through stencil stream `read_stream`, with the read address on the address stream `read_addr_stream` determining the read location in that bank. Meanwhile, data in the next data block is loaded to the other bank throughput stencil stream `write_stream`, with the write address on the address stream `write_addr_stream` determining the write location. `write_bound` and `read_bound` sets the number of cycles to write and read double buffer.
- `systolicarray(in_stream, w_stream, psum_stream, out_stream, [unrolled_dim1, unrolled_dim2], [array_size1, array_size2])` specifies a systolic array that computes a CONV layer or FC layer by taking a ifmap stencil stream `in_stream` and a weight stencil stream `w_stream` to update partial sums from `psum_stream` and generate ofmap stencil stream `out_stream`. `unrolled_dim1` and `unrolled_dim2` are a list of the loops that spatially unrolled and mapped on to the two dimensions of systolic array respectively. The systolic array size is `array_size1` $\times$ `array_size2`, which must be constant integers.

Using the above double buffer and systolic array abstraction, we can construct a DNN accelerator from the systolic architecture template in Figure 3.1(b). Figure 4.3 presents the version which pipelines the stencil stream. In the memory hierarchy, we allocate double buffers for ifmap, weight and ofmap. The ifmap double buffer and weight double buffer use `write_streams` to fetch stencils of ifmap pixels and weight coefficients in the next data block. At the same time, these double buffers transfer the stencils in the current data blocks to their `read_streams` for systolic array to read and compute. The systolic array generates output stencils of pixels and pushes to `write_stream`, which writes the data to ofmap double buffer. The address streams (`write_addr_stream` and `read_addr_stream`) are used to determine the write and read location of the double buffers.

Listing 4.4 presents a piece of extended IR program to describe the accelerator shown in Figure 4.3, mapping a CONV layer as an example. The generated output stencil stream is declared first, followed by declaring the internal streams include the data stencil streams and address streams for communication between double buffers and systolic array. The input stencil streams that fetch ifmaps and weights are not declared here as they are defined outside this scope, as are the write address stream for ifmaps and weights and the read address stream for ofmaps. The tile_(x,y,k) loop

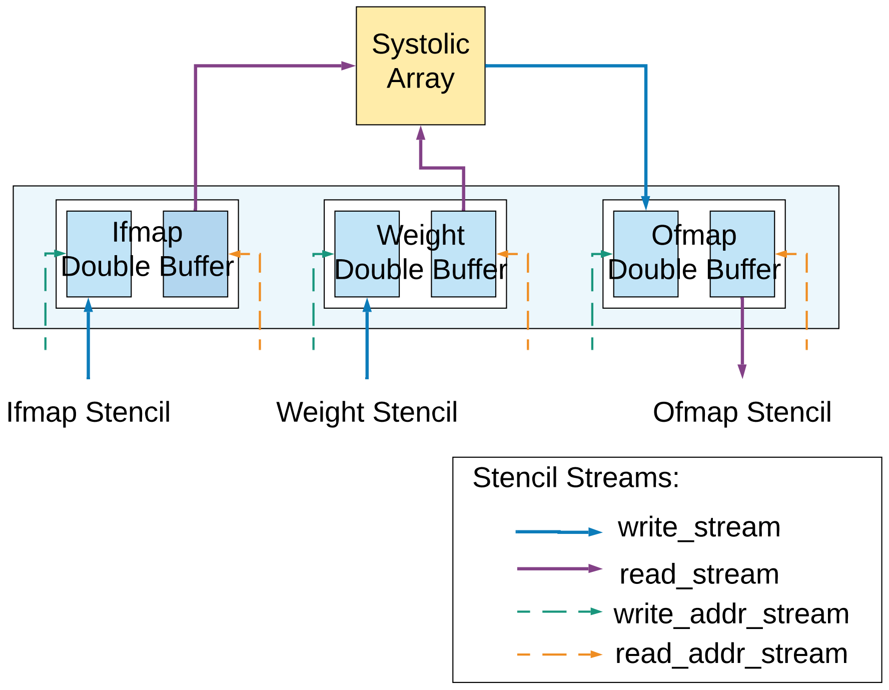


Figure 4.3: Block diagram of DNN accelerator design using stencil stream interface.

nests are the outer loops created by loop blocking. Using loop blocking, we break both the x- and y- dimension into 4 regions respectively, and k-dimension into 8 regions. This divides the original $128 \times 128 \times 512$ ofmap into tiles that are $32 \times 32 \times 64$. Before calling each `doublebuffer` function, the loop nests of the inner loops are utilized to calculate the address that are pushed to address streams, as the read address is determined by the execution order of the current compute kernel. Since the input channel dimension is not split in this example, we initialize the partial sum stencils to be 0, and send the initialized stencil stream to the systolic array. The function calls `doublebuffer` and `systolicarray` correspond to the double buffer and systolic array in the accelerator, respectively. We assume the address on the address streams are just scalar values. The size of data stencil on the stencil streams and the arguments used in the function calls can be derived by the Halide compiler analysis, which we will introduce more in the next Section.

To create DNN accelerator designs from the reduction tree architecture template in 3.1(c), we can replace the systolic array intrinsic function call in the IR program in Listing 4.4 with a unrolled loop nests of the compute kernel.

```
// Define generated stencil stream
def_stream(ofmap_stream, int, [])
// Define internal stencil streams for transferring data
def_stream(ifmap_stream, int, [])
def_stream(weight_stream, int, [])
def_stream(psum_stream, int, [])
def_stream(in_ofmap_stream, int, []
// Define address streams
def_stream(ifmap_rd_addr_stream, int, [1, 1])
def_stream(weight_rd_addr_stream, int, [1, 1])
def_stream(ofmap_wt_addr_stream, int, [1, 1])

// Loop iterate over different blocks
for (tile_y, 0, 4)
  for (tile_x, 0, 4)
    for (tile_k, 0, 8)
      // Generate read address streams for ifmap double buffer
      for (...)
        push(ifmap_rd_addr, ifmap_rd_addr_stream)
      // Instantiate double buffer for ifmap
      double_buffer(in_ifmap_stream, ifmap_stream, ifmap_wt_addr_stream, ifmap_rd_addr_stream,
                    ifmap_write_bound, ifmap_read_bound, 34*34*32)
      // Generate read address streams for weight double buffer
      for (...)
        push(weight_rd_addr, weight_rd_addr_stream)
      // Instantiate double buffer for weight double buffer
      double_buffer(in_weight_stream, weight_stream, weight_wt_addr_stream,
     weight_rd_addr_stream,
                    weight_write_bound, weight_read_bound, 32*64*3*3)
      // Initialize psum to 0
      for (...)
        push(0, psum_stream)
      // Instantiate 4 X 4 systolic array by unrolling C and K dimensions
      systolic_array(ifmap_stream, weight_stream, psum_stream, in_ofmap_stream, [[c], [k]], [4,
     4])
      // Generate write address streams for ofmap double buffer
      for (...)
        push(ofmap_wt_addr, ofmap_wt_addr_stream)
      // Instantiate double buffer for output
      double_buffer(in_ofmap_stream, ofmap_stream, ofmap_wt_addr_stream, ofmap_rd_addr_stream,
                    ofmap_write_bound, ofmap_read_bound, 32*32*64)
```

Listing 4.4: The extended IR program for a CONV layer

One issue of using the Halide IR to describe the DNN accelerator is the difficulty of reusing hardware blocks. As a functional language, each function in the Halide algorithm is lowered to a piece of IR program (similar to the IR shown in Listing 4.4), and each piece of IR program generates dedicated hardware blocks. As a result, specifying a DNN algorithm in Halide by having one Halide function for each layer leads to the inefficient design that have dedicated hardware modules for each individual layer. To avoid this area overhead and resource under-utilization, in our current design, we only map one layer on the hardware each time. We also have experimented with parametrizing the hardware blocks to support running multiple layers with different shapes. For instance, by parametrizing the input Halide algorithm, certain parameters such as the loop bound of the outer loops (`tile_y`, `tile_x` and `tile_k`) in the generated IR program in Listing 4.4 becomes variables. In hardware, there variables are generated as configurable registers, thereby allowing us to configure the hardware at runtime. However, generating hardware from parametrized Halide algorithms still requires the programmer to chose the set of parameters for reconfiguration, and also gives rise to extra complexity for the compiler design, thus we leave it as a future work.

4.3 Compiler Implementation

Our Halide compiler design is built on top of Pu's work [72], as presented in Figure 4.4. We add an analysis pass to extract parameter for the architecture template, and transformation passes to turn the code section marked for acceleration into the extended IR program suitable for High-Level Synthesis (HLS). After various compiler optimizations, the dataflow IR is passed to Vivado HLS [94] or Catapult HLS [32], which generate hardware designs targeted to FPGA and ASIC, respectively. [1]

Architecture Parameter Extraction: Our compiler generates DNN accelerators by instantiating and configuring architectural templates from a scheduled Halide program. The architectural template as presented in Figure 3.1(b) and Figure 3.1(c) are PE array (systolic array or reduction tree) with double buffers, communicating through stencil stream interfaces. The architecture parameters to be extracted include the parameters of each stencil stream, each double buffer, and each systolic array/reduction tree.

To extract the stencil sizes of each stencil stream in the template shown in Listing 4.4, we leverages the *bound inference* analysis in the existing Halide compiler [74]. Since the stencil stream interface is used to provide high bandwidth matching computation throughput, the stencil must serve all the data required by the computation kernel (layer in DNNs) at each cycle. Therefore, we can apply the bound inference analysis to compute the stencil size as the required data block sizes at the granularity of the unrolled loop. For example, if we unroll the input channel loop (c) by 4, and output channel loop (k) by 8. The stencil sizes of ifmap stencil stream and weight stencil stream are $[4]$ and $[4, 8]$ respectively.

The double buffer size can also be extracted by bound inference analysis. Each buffer size is the

[1]The current compiler implementation is deprecated, more information about the current developing system can be found in a follow-up work by Jeff Setter and Qiaoyi Liu et al, submitted to ASPLOS 2020. The source is available at https://github.com/StanfordAHA/Halide-to-Hardware

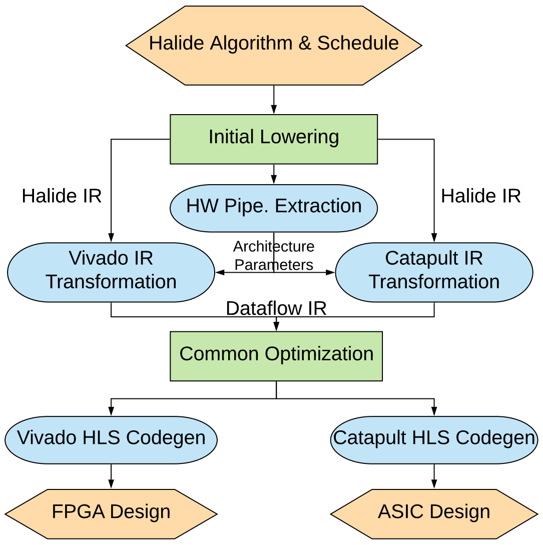


Figure 4.4: Compiler flow. Blue blocks are new, green blocks are existing Halide compilation passes.

required data block size at the granularity of the buffer stored level. For the example in Listing 4.1, both ifmap and weight double buffer are scheduled to be store at loop level `xo`, as `in()` is `compute_at` that loop. The write bound and read bound control the number of stencils write to and read from the stencil streams respectively. The write count is the data block size, and the read count equals to the total number of loop iterations in the compute kernel that read the double buffer. Both of them can be calculated based on the tiled loop sizes in the IR.

The parameters of systolic array include the unrolled dimensions and array size. The first one are just the unrolled loops in the scheduled program. The array size can be directly extracted from the unrolled factors. If using a reduction tree architecture template, we can skip extracting these parameters, as the generated IR program with unrolled loops already matches the HLS code structure to generate the hardware for reduction tree.

IR Transformation: The Halide compiler lowers a scheduled algorithm to the IR program with loop nests and storage allocations injected for each function. We add a new compilation pass to transform the Halide IR into the extended IR, using the previously extracted architectural template parameters. First, we insert `doublebuffer` intrinsic function calls before and after the computation kernel to buffer the inputs and outputs of each stage (Line 21, 27, 38 in Listing 4.4). These function calls replace the generated IR piece by `in` schedule primitive, which includes buffer allocations and loop nests of copying data. To generate the address streams of the double buffers, we also create the loop nests that compute the address and push to streams before each `doublebuffer` call (i.e. Line 18-19 in Listing 4.4). The loop nests are constructed from the loop nests of the computation kernel and the addresses of load or store operations to the original buffer. Finally, if the architecture is scheduled as systolic array, we replace the computation loop nests with the `systolicarray` function call (Line 33 in Listing 4.4).

Code Generation: We created two HLS code generators — a Vivado HLS code generator and a Catapult HLS code generator, targeting FPGA and ASIC backends respectively, The HLS code generator translates the accelerator portion of IR into HLS-synthesizable C++ code. It also inserts HLS directives (pragmas) to enable HLS tools applying hardware-oriented optimizations including loop pipelining and array partitioning. To reduce the complexity of the code generator, we developed a HLS-synthesizable C++ template libraries implementing an abstract double buffer and systolic array interfaces (Section 4.4). To create hardware instances, the code generator just emit function calls to the corresponding library.

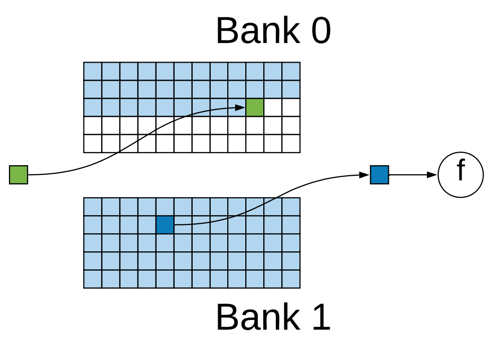
4.4 Hardware Generation

Our Halide compiler translates high-level Halide algorithms to hardware implementations based on the architectural template in Section 3.1. The template consists of common building blocks including double buffers and systolic arrays, which are useful for improving throughput and optimizing energy efficiency for DNN applications. This section introduces the abstraction definition and efficient hardware implementations of these blocks.

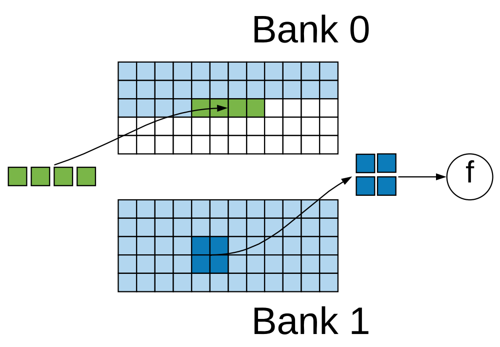
4.4.1 Double Buffer

Double buffers are the main on-chip memory in the DNN accelerator architecture. DNN algorithms often have extremely high locality: ifmaps are reused by multiple filters and all coefficients within a filter window; ofmaps are updated by multiple input channels and all ifmap pixels within a window; filters are reused by all pixels and batches. In the absence of buffers, these feature map pixels and weight coefficients have to be refetched multiple times, causing redundant main memory references and significant energy overhead. To overcome this issue, we can design a memory hierarchy, and block the computation to maximize the locality. With static knowledge of the access pattern of the algorithm, a double buffer allows fetching the next data block while computing on the current one. Therefore, the data communication latency can be overlapped, and in most cases hidden, by the computation.

Figure 4.5(a) presents a typical double buffer in a DNN algorithm. It is composed of two banks — bank0 and bank1. The pixels in light blue denotes the elements currently stored inside the banks. The pixel in green is the value from the next data block to be stored in bank 0, while the pixel highlighted in dark blue is the one currently read from bank 1 for computation. As depicted in this Figure, each bank is used to store a unique block of data, when bank 1 is serving data for the current computation, pixels from the new block are continuously writing to bank0. As long as there is sufficient computation to reuse the data in bank 1, the cycles for filling bank 0 with the next data



(a) A double buffer with single pixel input and output.



(b) A double buffer for 1 × 4 input and 2 × 2 output.

Figure 4.5: Double buffer with two banks to hide data communication latency.

block can be hided. When all the operations that use the block inside bank 1 have been performed, the two banks switch, with new data block writing to bank 1, and computation reading data from bank 0.

The model of a double buffer can be generalized for handling input and output of multiple elements, as shown in Figure 4.5(b). This is useful to support applying `unroll` primitives to schedule a function at a higher throughput rate. For example, if the kernel in Figure 4.5(a) is unrolled by a factor of 2 in both x- and y- dimensions, the realized computation units will consume stencils of 2 × 2 pixels. Similarly, if the produced data of a upstream kernel is written to the double buffer, when the upstream function is unrolled by a constant factor, the inputs to the double buffer are stencils of elements.

Unlike a line buffer for image processing algorithm, where the pixels/stencils are streamed to functional units in raster-scan order, the access patterns in double buffers can be more complicated. The addresses may not increase monotonically, instead it can traverse the data block inside the buffer in a way that maximize the data reuse. For example, data can be scanned within a subblock before moves to the next one, also known as Zigzag pattern. One of the cases that can also cause non-consecutive access is performing layout transformation. For instance, when the double buffers are allocated between two kernels, with the downstream kernel using a different memory layout from the upstream kernel, the double buffer has to shuffle the data internally to serve the data with the required sequence. Therefore, double buffer needs to be designed to support flexible access patterns in addition to hiding communication latency and capturing locality.

Figure 4.6 presents the block diagram of a double buffer design for outputting 1 × 4 stencils and inputting 1 × 4 stencils. The hardware is made up of two banks and a control unit. The control unit manages the read and write operations alternating between the two banks. Each bank provides sufficient storage to buffer a data block and exploit the data reuse, with the storage size being not

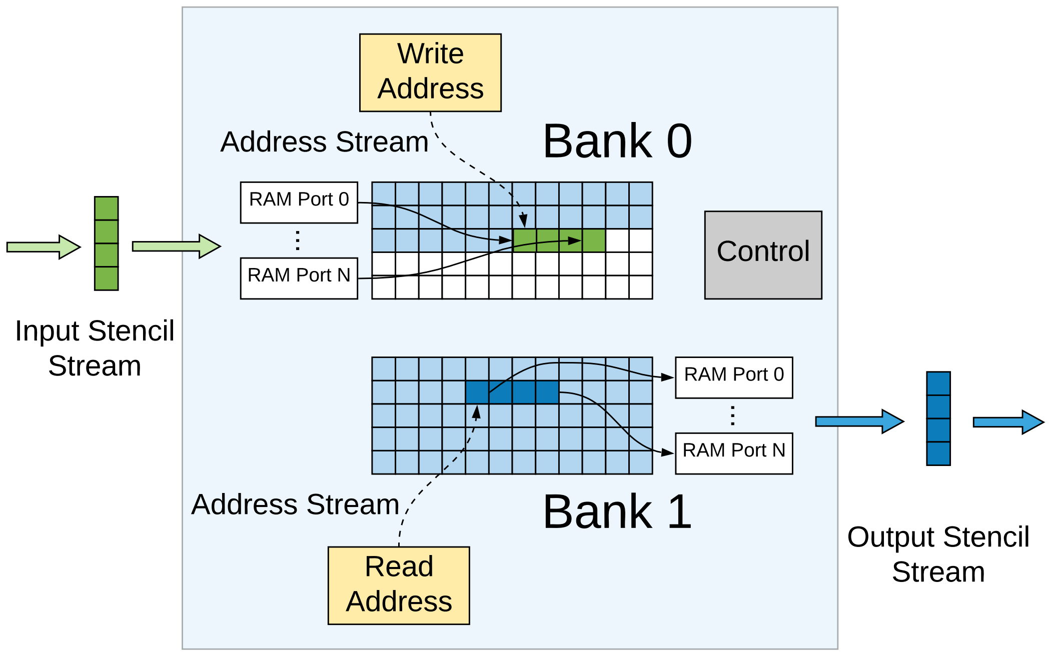


Figure 4.6: Block diagram of the double buffer design.

less than the block size. When used for storing high-dimensional array of data, the array is flattened into 1D.

To provide sufficient bandwidth for reading and storing data, it is instantiated using a RAM with a wide access port (or sometimes a multi-port RAM), which can be implemented using BRAM blocks in FPGA/CGRA or SRAM blocks in ASIC. The data width of the RAMs equals the aggregate word width of all the pixels in the stencil, which allows a full stencil to be read or written into the buffer within one cycle. For the example in Figure 4.6, each bank is a four-element-word wide. If given the maximum port width of a BRAM or SRAM block $port\_width$, and the data width of each banking $data\_width$, the required number of BRAM or SRAM blocks (N in the figure) should be no less than $data\_width/port\_width$ otherwise the downstream kernels may be stalled.

The interface to the buffer consists of two stencil streams for input and output data, and two stencil streams for read and write addresses, as illustrated in Figure 4.6. The address streams are provided to support flexible memory access patterns. The double buffer have three working modes: only write a bank (write-only), only read a bank (read-only), and write and read corresponding banks concurrently (write-read). Taking the last mode as an example, each cycle, a new input stencil and write address is obtained from the input stencil stream and write address stream respectively (assume not empty), then the input stencil is written to the location determined by the write address. Meanwhile, a new stencil is read from the read bank from the location determined by the read address fetched from the read address stream. The data stencil in the stencil stream can be high-dimensional, whose size and dimension are the same as the ones of the stencils that stored in the buffer. For simplicity, the address stream is just a stream of integers, since the address space has been flattened into 1D. The control unit utilizes the counters and FSMs to control the mode of the double buffers, and the total number of write and read operations issued to the buffer. The abstraction of a double buffer can be defined as a C++ class template, as presented in Listing 4.5.

```
template<typename T, int BUFFER_SIZE, int SIZE_1, int SIZE_2, ..., int SIZE_N>
class Doublebuffer{
    // Allocate two banks for double buffer, each has size BUFFER_SIZE
    stencil<T, SIZE_1, SIZE_2, ..., SIZE_N> bank0[BUFFER_SIZE];
    stencil<T, SIZE_1, SIZE_2, ..., SIZE_N> bank1[BUFFER_SIZE];
    bool read_bank0 = false;
    int counter = 0;

    // Top level function that switch banks at each iteration
    void execute_buffer(stream<stencil<T, SIZE_1, SIZE_2, ..., SIZE_N>> &write_stream,
              stream<stencil<T, SIZE_1, SIZE_2, ..., SIZE_N>> &read_stream,
              stream<int> &write_addr,
              stream<int> &read_addr,
              int write_bound,
              int read_bound);
}
```

Listing 4.5: Double buffer library implementation using HLS. It is instantiated and configured by generated code from Halide.

The class template parameters are the buffer size (`BUFFER_SIZE`), the stencil extent (`SIZE_i`) of the stencil stream at each of the N dimensions and the data type of pixels (`T`). The function `execute_buffer` takes an input and an output stencil stream (`stream<stencil<...>>` objects), an read and an write address stream (`stream<int>` objects) and an read and an write operation count (`write_bound` and `read_bound`) as arguments. These bounds are used to configures the number of stencils we write to or read from the double buffers. Similar to the template designed by Pu et al. [72], the `stencil` class represents a multi-dimensional array of elements. A `stream` object denotes a FIFO interface, supporting push and pop operations for the caller. Note, we constrain the input and output stencil to be of the same size and dimension, so that the interface is still consistent after bank switching.

Inside the buffer, we explicitly create two arrays as the two banks, each of size `BUFFER_SIZE`. This parameter is derived from the extended Halide compiler to be sufficiently large to buffer a scheduled data block, as introduced in Section 4.3. Each element in the array is a N dimensional `stencil` object with the same type and dimension as the ones in the stencil stream. The flag `read_bank0` is utilized to determine the current bank to read. The counter `counter` takes track of the number of data blocks that have been loaded to the double buffer, which is used to choose the working mode

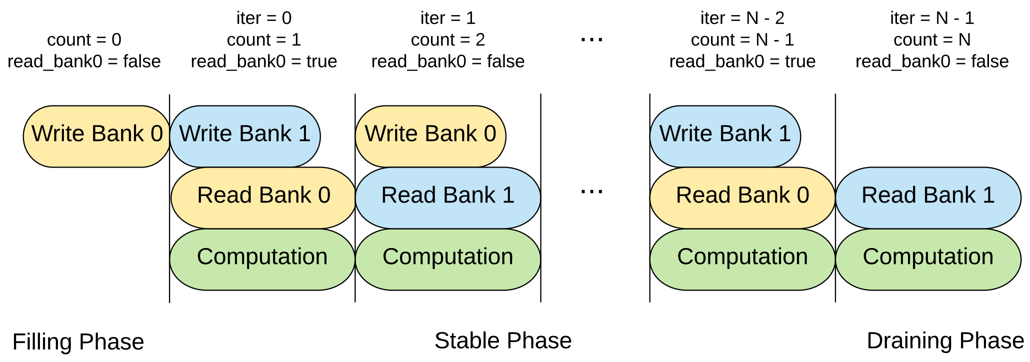


Figure 4.7: Timing graph of a double buffer design.

of the double buffer. The flag and counter will be used later in Listing 4.6.

This template interface matches the definition of the `doublebuffer` function in the dataflow IR (see Section 4.2). Thus, to instantiate a double buffer for each `doublebuffer` call in the IR, the compiler just emits an instantiation of this class template and a call to the `execute_buffer` function template during code generation. This template interface is implemented in a HLS C++ template library, which can be used to create instances of double buffers by leveraging the C++ front-end in the HLS tool. Due to the restricts of the C++ template, the maximum dimension N has to be specified ahead-of-time. We use $N = 4$, which is sufficient for most of the data dimensions used in DNN algorithms.

The control unit in Figure 4.6 is made up of FSMs that utilizes flags and loop counters to manage the bank switching. Before introducing the mechanisms of the control logic, we first look at the timing graph of data transfer and computation, using the example in Figure 4.7. At first, one block of data is written to bank 0. Since we start to fill data into the pipeline, we name the current phase as filling phase. Following the filling phase is the stable phase, during this period, at the first iteration, the data block previously stored in bank 0 is send to the computation units for processing, while the next block is transferred to bank 1. At the next iteration, the data block is bank 1 is ready for computation, while data in bank 0 have already been fully used, thus the data in bank 0 is replaced with the new data block. For the rest iterations in the stable phase, it repeats this process for data transfer and computation. In the end, the last data block is ready for computation in bank 1, with no new data block coming, we drain the pipeline with reading bank 1, and we call the last stage draining phase. Shown in this example, the computation takes longer cycles than data transfer, thus no stalls or bubbles in the pipeline.

We implement this mechanisms in the function template as a member function of the `Doublebuffer` class, which is presented in Listing 4.6. The function parameters are the same as the ones of the class template, which are the data type, buffer size and the stencil extents. This function is implemented as a HLS C++ function template, with taking the input and output stencil steams, the read and write address streams and the number of stencils to write to and read from the buffer as arguments.

```cpp
template<typename T, int BUFFER_SIZE, int SIZE_1, int SIZE_2, ..., int SIZE_N>
void Doublebuffer<T, BUFFER_SIZE, SIZE_1, SIZE_2, ..., SIZE_N>::execute_buffer(
            stream<stencil<T, SIZE_1, SIZE_2, ..., SIZE_N>> &write_stream,
            stream<stencil<T, SIZE_1, SIZE_2, ..., SIZE_N>> &read_stream,
            stream<int> &write_addr,
            stream<int> &read_addr,
            ...) {
   if(read_bank0) {
        // Read data from bank0 to read_stream using the address in the stream read_addr
        // The number of elements to read is read_bound
        read_bank(bank0, read_addr, read_stream, read_bound);
        // Write data from the write_stream to bank1 using the address in the write_addr
        // The number of elements to write is write_bound
        write_bank(write_stream, write_addr, bank1, write_bound);
   }else {
        if(count != 0) {
            // Read data from bank1 to read_stream using the address in the stream read_addr
            read_bank(bank1, read_addr, read_stream, read_bound);
        }
            // Write data from the write_stream to bank0 using the address in the write_addr
            write_bank(write_stream, write_addr, bank0, write_bound);
   }
   read_bank0 = 1 - read_bank0;
   count++;
   }
}
```

Listing 4.6: Double buffer function in HLS. It adopts software pipeline technique to switch the read and write banks at each iteration.

The bank switching mechanism in realized by using software pipeline technique. This technique skews the iterations of the pipeline, so that bank reading and writing operations can be performed concurrently at each iteration. For example, as demonstrated in Figure 4.7, the bank writing operation starts before the first iteration ($iter = 0$), and finishes before the last iteration ($iter = N - 1$). As a result, at iteration i, the data block $i + 1$ is written to one bank, while the data block i from another bank is read to computation. To implement loop skewing, we start by realizing the filling phase by setting the double buffer to the write-only mode. We check the value of `counter` to determine whether it is the filling phase of the pipeline. If so, we only call the `write_bank` function during that iteration, which fills one bank with the data from the input stencil stream `input_stream`. During the stable phase, the `execute_buffer` function is called at each loop iteration to concurrently write data block $i+1$ by the `write_bank` function and read data block i by the `read_bank` function.

The flag `read_bank0` is utilized to determine the bank indices that are passed to `read_bank` and the `write_bank` functions, which alternates between 0 and 1 every iteration. At the last iteration, which is the draining phase, the `read_bank` function is called to copy the data from the last data block to the output stencil stream `output_stream`. Note, even though we also call the `write_bank` function at the last iteration, when designed properly it take no effect: no elements are on the write stencil stream `write_stream` and reading from an empty stream is essentially a noop.

To realize the process in the timing graph in Figure 4.7, this `execute_buffer` function is used to switch banks, and is designed to be called every loop iteration in Figure 4.7. Note, the loop iterators are implemented outside the `execute_buffer` function, rather than inside. They are used to create the read and write the address streams and data stencil streams passed to this function. Thus the only control in the `execute_buffer` function to switch read and write banks which is easy to implement. Decoupling address generation from the double buffer also improves the flexibility of the buffer, enabling it to support various access patterns.

4.4.2 Systolic Array

DNN algorithms have massive parallelism that can be exploited to improve throughput, specifically the seven dimensions in Algorithm 1 can all be parallelized, as introduced in Section 3.1.2. To better utilize the parallelism, we can design a computation template that utilizes sufficient PEs to spatially unroll those dimensions. However, simply creating a sets of PEs on top of the memory hierarchy doesn't solve all the problems, as spatially mapping algorithms also result in the overhead of data communication and/or data replication. For example, spatially unrolling the input channel dimension (C) requires frequent data communication with memory to update the partial sums, spatially unrolling the output channel dimension (K) demands either broadcasting the weights to all PEs or storing a copy of weight block inside each PE. Thus, the computation template has to be designed with the capability of exploiting parallelism and alleviating the energy cost caused by the data communication and replication

A systolic array was initially proposed to efficiently perform dense linear algebra computation. Noticing the similarity of the computation pattern between DNN and linear algebra, it was widely use in prior works as the computation architecture template for DNN accelerators. A systolic array is a collection of PEs, usually containing MAC units, connected in a homogeneous network, typically arranged in a 2D grid, as shown in Figure 3.1(b). The set of the PEs provides sufficient computation resource to spatially unroll the algorithms, thereby exploiting parallelism and improving throughput. The homogeneous mesh-based network allows data transferring among neighbor PEs, substituting the expensive memory references with cheap and direct communication on the interconnects.

Figure 4.8 depicts an example of systolic design for DNN algorithms. It is made up of 4×4 PEs, arranged in a 2D grid. Each PE contains one MAC with a local register file to perform computation. The PEs are connected is a way that each PE can receive data from its adjacent PEs on the left-side and above, and send data to its adjacent PEs on the right-side and below. There are many other ways of designing the connections of the PEs, the interconnect in this figure is just one example. To meet efficiency and performance requirements, the network of systolic array needs to be redesigned based on the computation pattern and the schedule of the computation, as was explored in Eyeriss V2 [14] and Cong et la. [20].

In this example, we implement the dataflow $C \mid K$, unrolling input and output channel dimensions vertically and horizontally, respectively. This schedule essentially transforms the computation of CONV layer into a dense matrix multiplication, and spatially maps it onto the systolic array. As illustrated in Figure 3.2(a) and Figure 4.2(b), weight coefficients for different filters are distributed in different columns, and the ones for different input channels are stored in different rows. Stencils of ifmaps are sent to the first column of PEs, with the input pixel at stencil location i being transferred to the first PE at row i. The input data is propagated from the left-most PEs to the right-most PEs using systolic transfer, with each row transferring a different ifmaps channel. The partial sums obtained from different input channels are accumulated vertically across the PEs. The partial sums to be updated in the psum stencils are transferred to the first row, and the updated ones in output stencils are sent out from the bottom row. The produced output stencils consist of outputs in different output channels, computed by different columns with the corresponding filters. The output at stencil location i is produced by the bottom PE in column i.

To realize the correct functionality with this schedule, the input and psum stencils need to arrive at the right timing, and the data in the output stencils need to be received at the right timing as well. For example, assume the weights have been preloaded into the corresponding PEs. To perform the partial sum accumulation, at the first cycle, only PE(0,0) starts computation, multiplying the input data from the first channel with the first filter. At the second cycle, the input pixel from PE(0,0) can be propagated to PE(0,1), and the partial sum calculated by PE(0,0) is ready to be transferred to PE(1,0) for accumulation, thus PE(0,1) and PE(1,0) also starts processing. At the next cycle, PEs on the next diagonal line received the required data for computation and starts working. Repeating this process, we observe the input stencils and psum stencils should come with a diagonal wave-front. Specifically, the PEs in the first row or first column receive data with no delays, the PEs at the second row or second column wait one additionally cycle to receive data, the PEs at the third row or column have two cycle delays, and so on. Similarly, the bottom PE in the second column sends out the output one cycle after the bottom PE in the first column, the one in the third column have two cycle delays and so on.

Even if the input and output data from the same stencil needs to be read from or written to memory simultaneously, due to the wave-front characteristic, they don't arrive or come out from the PE array at the same time. To overcome this issue, it is possible to build a FSM inside each PE to control the data communication and start of operations. However, adding logic to the PE creates redundant control logic. Instead, as shown in Figure 4.8, we create FIFOs around the PE arrays to add the proper delays for incoming and outgoing data. With a diagonal shape of FIFOs, the input and output data can be realigned within each stencil, so the memories only see unskewed data.

The interface to the systolic array are the three stencil streams for input, weight, and output data, as presented in Figure 4.8. The stencil streams connect with the outside memory. The weights can be preloaded from the weight stencil stream to the register file inside each PE before computation starts. But in order to hide the weight transfer latency, we load the weights while fetching input and partial sums. Once the required weights have been stored inside PEs for reuse, we stop popping new weight stencils from the stencil stream. Once all the data has been loaded, on each cycle, a new input stencil and psum stencil are obtained from the input stencil stream and psum stencil stream and sent to the FIFOs. After certain number of cycles of computation to get the first output stencil computed and passed through the output FIFOs, a new output is pushed to the output stencil stream every cycle.

The stream interfaces to the systolic array must be sufficiently wide to satisfy the bandwidth required by the computation. Since the stencil sizes of input, psum, weight and output stencils determine the width of the stream interface, they can be set based on the dimensions of the systolic array to achieve high PE utilization. When a single ifmap pixel and weight is required by each PE, the input stencil is `stencil<T, ROW_NUM>`, with `ROW_NUM` being the row count. The output stencil is `stencil<T, COL_NUM>`, with `COL_NUM` being the column count. We constrain the psum stencil and output stencil to be of the same dimension and size, as the output stencil is an update on psum stencil. For our design, weight stencils are loaded row by row, from the top to the bottom one, so weight stencil is `stencil<T, COL_NUM>`. To support higher throughput rate with higher required bandwidth by the computation, we can set the input, psum, weight and output stencils as high-dimensional stencils. For example, the input stencil becomes `stencil<T, SIZE_1, SIZE_2, ... SIZE_N, ROW_COUNT>`. In this case, each transfer between adjacent PEs is a packed block of elements, expressed as `stencil<T, SIZE_1, SIZE_2, ... SIZE_N`.

The function template `systolic_array` in Listing 4.7 takes an input, a weight, a psum and an output stencil stream (`stream<stencil<T, ...>>` objects) as arguments. The parameters such as `ROW_NUM` and `COL_NUM` can be derived from the unrolled factors by the extended Halide compiler. Similar to double buffer template, this template interface is implemented as a HLS C++ template library, which can be used to synthesize systolic arrays by a HLS tool. To provide the data to the systolic array, we connect the output stencil streams of double buffers to the input, weight and psum stencil streams of the systolic array.

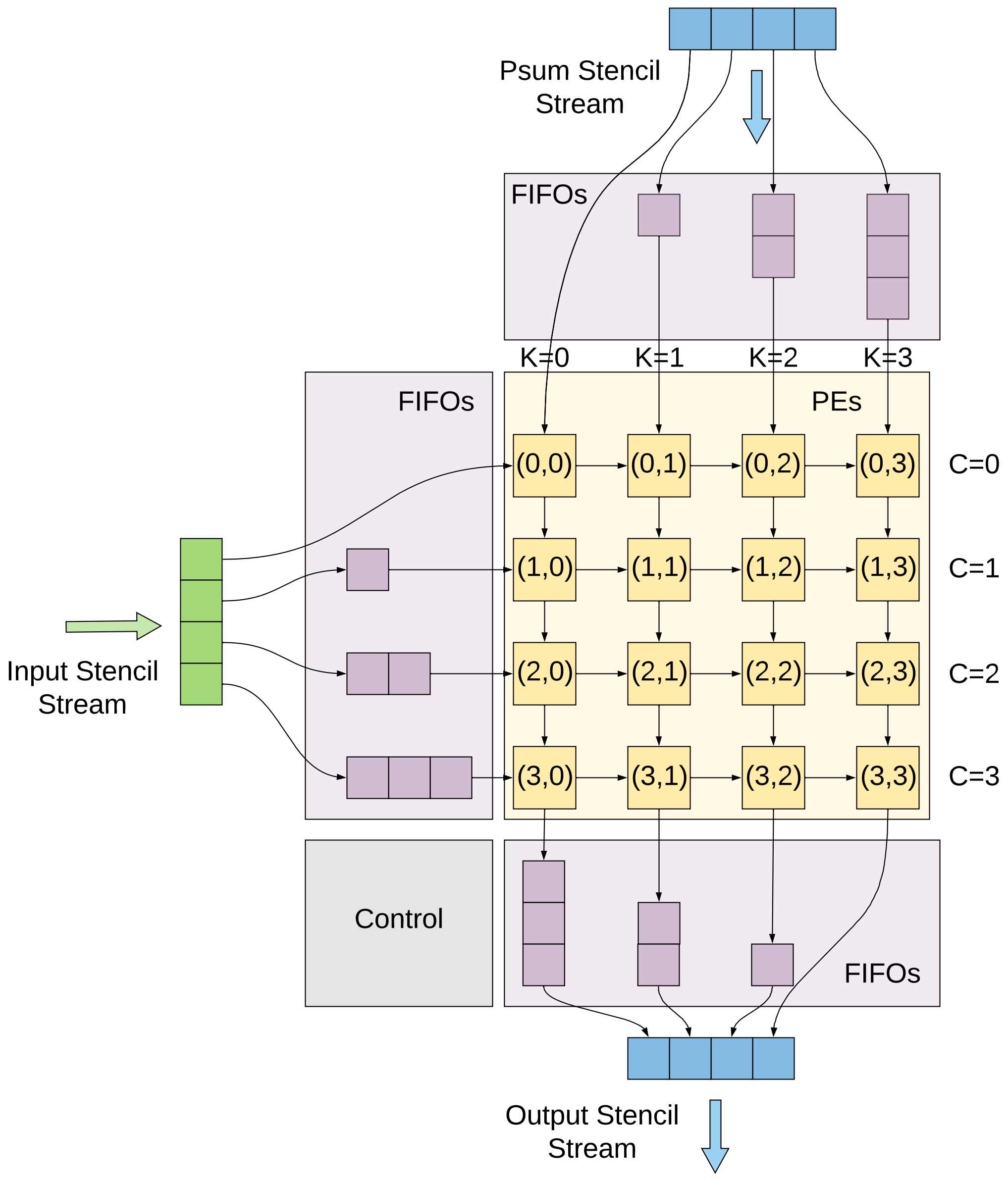


Figure 4.8: Block diagram of the systolic array design. Ifmap pixels are transferred from first column to last column, weights are stored and reused inside each PE, partial sums are accumulated vertically from the top row to the bottom row. A diagonal shape of FIFOs are added around the PE array to add appropriate delays for data coming in and out of the systolic array.

```
template<typename T, ROW_NUM, COL_NUM, ...>
void systolic_array(stream<stencil<T, ROW_NUM>> &input_stream,
                    stream<stencil<T, COL_NUM>> &weight_stream
                    stream<stencil<T, COL_NUM>> &psum_stream,
                    stream<stencil<T, COL_NUM>> &output_stream){

    ProcessingElement<T, ... > PEs[ROW_NUM][COL_NUM];
    //Input FIFOs to ensure the input data arrive at the right timing
    FIFOs<stencil<T, ROW_NUM>> input_fifos;
    //Psum FIFOs to ensure the partial sum data arrive at the right timing
    FIFOs<stencil<T, COL_NUM>> psum_fifos;
    //Output FIFOs to ensure the output data is send out at the right timing
    FIFOs<stencil<T,  COL_NUM>> output_fifos;
    T row_reg[ROW_NUM][COL_NUM];
    T col_reg[ROW_NUM][COL_NUM];

    for( ... )
      ...
      input_fifos.call(input_stream, row_reg);
      psum_fifos.call(psum_stream, col_reg);
      //Instantiate the PE array ARRAY_SIZE_1 x ARRAY_SIZE_2
      #pragma hls_unroll
      for(int j = 0; j < ROW_NUM; j++) { // Row
        for(int i = 0; i < COL_NUM; i++) { // Col
          PEs[j][i].call(row_reg[j][i], col_reg[j][i], row_reg[j][i+1], col_reg[j+1][i], w);
        }
      }
      output_fifos.call(col_reg, output_stream);

}
```

Listing 4.7: An example of generated systolic array in HLS. It is used to instantiate and configure systolic array by the generated code from Halide for systolic array architecture in Figure 4.8.

Inside the systolic array, we create a 2D array of a `ProcessingElement` object, and `FIFOs` objects for adding delays to adjust timing. Figure 4.9 illustrates the PE architecture. The green registers are the row registers for propagating input stencils horizontally, the blue registers are the column registers for accumulating partial sums vertically. Inside each PE, there is local storage, typically register files for reusing data locally. The control unit, made up of FSMs and muxes, manages the data communication and the operation of the execution unit.The execution units performs the required computation, which is just one MAC unit in our design.

To build the homogeneous network in the systolic array, we register the inputs of each PE as shown in Figure 4.9, and register the output of the entire array. This makes the cycle time equals the computational delay of the PE plus the local communication delay. These registers give rise

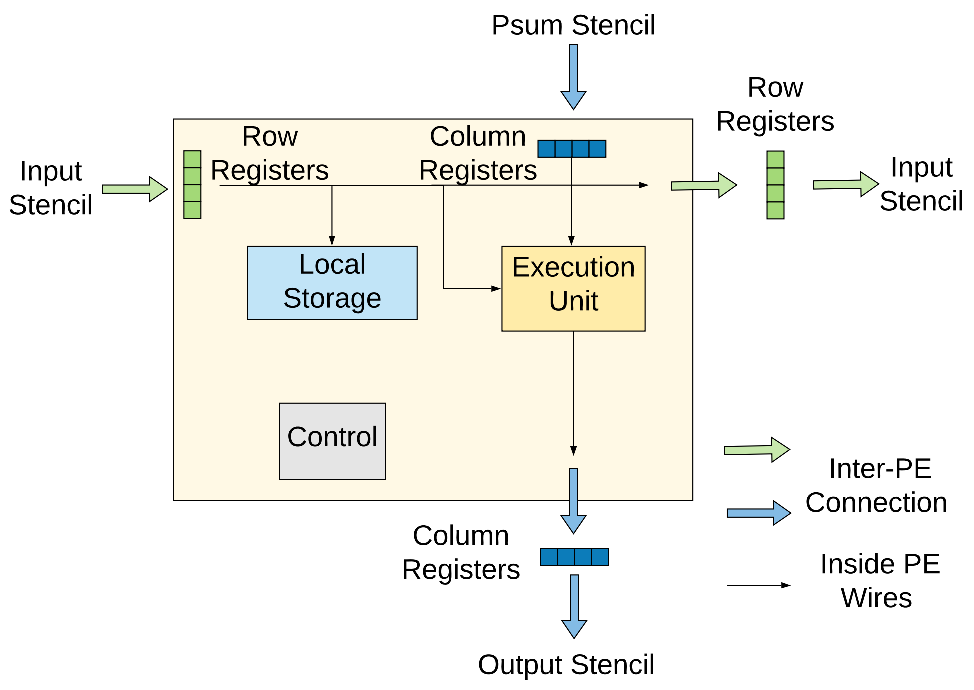


Figure 4.9: Block diagram of the PE architecture design.

to the one cycle delay as data moves from PE to PE that is associated with systolic arrays. For the example in Listing 4.7, PE(1, 1) receives input data from its input register (`row_reg[1][1]`), and drives this on its row outputs on the right hand side of the PE. On the next cycle, this data is clocked into (`row_reg[1][2]`). Similarly, to accumulate the output, PE(1,1) updates the psum data that was loaded into`col_reg[1][1]`, and drives the accumulated value on the outputs at the bottom of the PE. On the next cycle this value is loaded into`col_reg[2][1]` for PE(2,1) to continue accumulation. The current implementation connects every nearest-neighbor PE pair by the setting the dependency between reg[j][i] and reg[j][i+1], and between reg[j][i] and reg[j+1][i]. By setting the dependency between other register pairs, i.e. reg[j][i] and reg[j][i+2], it is possible to build other interconnect implementations.

4.5 Summary

Leveraging Halide compact scheduling language, we can precisely describe the design choices in the space as schedules of a program, and generate accelerators with any possible micro-architectures and schedules. Our extended Halide system enables programmers to nicely describe DNN algorithms and generate efficient accelerator implementations based on the double buffer based architectural templates. Leveraging the Halide compact scheduling language, it also allows us to concisely express and fairly compare various design choices about the underlying hardware and schedules. The next chapter will move on to create an analytical optimization framework that further accelerates the design space exploration.

Chapter 5

Optimizer and Results

Developing efficient hardware acceleration for DNNs is challenging, since the optimal micro-architecture depends on the schedule, so it is critical to optimize them together. The Halide language already provides the ability to express different micro-architecture parameters like loop blocking and dataflow. By extending Halide system as proposed in Chapter 4, accelerators with different parameters can be generated for energy efficiency and performance comparison. Since running through the full design evaluation pipeline, including RTL generation, synthesis, validation, place-and-route, etc., can take take a day or more of computation, exploring the huge design space introduced in Chapter 3 is difficult. Therefore, it is necessary to develop a computationally simple performance and cost model, and systematical approach that can efficiently consider all trade-offs and constraints, to fully analyze the entire space to determine the optimal one.

In this chapter, we first describe our computationally simple performance and energy modeling in Section 5.1.1 and Section 5.1.2. Then we validate these model against prior work and synthesized designs in Section 5.1.3. Employing these models, we propose a basic optimization framework in Section 5.2 which uses exhaustive search to systematically study the design space. With user specifying layer parameters and hardware configurations, it can analyze the impact of each dimension of the 3D design space individually, and can search for the optimal schedule and/or hardware resources.

By using the basic optimizer, Section 5.3 discusses the insights we obtained about the space. We observe many dataflows achieve similar energy efficiency, while optimizing hardware resource allocation and loop blocking can significantly improve the energy efficiency. Leveraging the heuristics, we further create an efficient optimizer in Section 5.3.2, as an improvement over the basic optimizer. It prunes the huge space for faster exploration and optimization. Using the efficient optimizer, we jointly optimize the hardware resource and schedule for various networks for better energy efficiency or throughput.

5.1 Energy and Performance Models

A systematic analysis and optimization framework requires a performance and energy cost model that can efficiently and accurately evaluate a given design candidate. Since both performance and energy cost depend on how data are blocked and assigned at each level, we first look at data blocking and assignment before introducing the performance and memory energy cost estimation.

```
// memory level 2
Alloc I[32][8][34][34]  // ifmap buffer [B0*B1*B2][C0*C1*C2][(Y0*Y1*Y2+FY-1)][(X0*X1*X2+FX-1)]
Alloc O[32][32][32][32]  // ofmap buffer [B0*B1*B2][K0*K1*K2][Y0*Y1*Y2][X0*X1*X2]
Alloc W[32][8][3][3]  // weight buffer [K0*K1*K2][C0*C1*C2][FY][FX]
for (int b2 = 0; b2 < 32; b2++)
  for (int k2 = 0; k2 < 8; k2++)
    for (int c2 = 0; c2 < 4; c2++)
// memory level 1
      Alloc I1[1][2][34][34]  // ifmap buffer [B0*B1][C0*C1][(Y0*Y1+FY-1)][(X0*X1+FX-1)]
      Alloc O1[1][4][32][32]  // ofmap buffer [B0*B1][K0*K1][Y0*Y1][X0*X1]
      Alloc W1[4][2][3][3]  // weight buffer [K0*K1][C0*C1][FY][FX]
      for (int y1 = 0; y1 < 2; y1++)
        for (int x1 = 0; x1 < 2; x1++)
// memory level 0
          Alloc I0[1][2][18][18]  // ifmap buffer [B0][C0][(Y0+FY-1)][(X0+FX-1)]
          Alloc O0[1][4][16][16]  // ofmap buffer [B0][K0][Y0][X0]
          Alloc W0[4][2][3][3]  // weight buffer [K0][C0][FY][FX]
          for (int k0 = 0; k0 < 4; k0++)
            for (int c0 = 0; c0 < 2; c0++)
              for (int y0 = 0; y0 < 16; y0++)
                for (int x0 = 0; x0 < 16; x0++)
                  for (int fy = 0; fy < 3; fy++)
                    for (int fx = 0; fx < 3; fx++)
                      O0[b0][k0][y0][x0] += I0[b0][c0][y0+fy][x0+fx] * W0[k0][c0][fy][fx]
```

Listing 5.1: An example schedule with data blocking for a CONV layer.

Given an L-level hierarchy, we define level 0 is the level closest to processor, while level $L-1$ is DRAM. Since access cost increases with increasing memory level, ifmap, ofmap and weights are iteratively blocked, the blocks at level i are partitioned into subsets. A subset is temporarily stored at level $i-1$ for more efficient reuse. Hence, the block size decreases from Level $L-1$ to 0, as demonstrated in Listing 5.1, which is an example of Algorithm 2 with one choice of tile sizes. This example schedule partitions both ifmaps and ofmaps along the batch b dimension, each ifmap and ofmap are also blocked along the channel dimension (c and k) to store 2 channels of ifmap and 4 channels of ofmap respectively inside buffers at level 1. The weight coefficients are also blocked into smaller blocks along c and k dimensions. Next, it continues to split ifmaps and ofmaps inside the buffer at level 1 into 4 tiles of size 18×18 and 16×16 within each channel plane, the subblocks of ifmaps and ofmaps are stored at level 0. To make the memory system more flexible, we conservatively

Table 5.1: The dimensions of each data block (set D), and loops reuse the data block (set V)

Data	**Dimensions**(D)	**Loops with reuse**(V)
ifmap	X, Y, C, B	f_x, f_y, k
ofmap	X, Y, K, B	f_x, f_y, c
weight	F_X, F_Y, C, K	x, y, b

assume the blocks at level i always include all content of the blocks at levels j $(0 \leq j < i)$. Frequently, scratch pad is utilized as the storage for specialized accelerators, but if this policy is used in a cache, we would call it an inclusive cache. Due to this policy, even the weight block is not partitioned at level 1, the same block will be stored at both level 1 and level 0.

For each data type, given a schedule, the data block size s_i can be directly calculated as the product of the sizes of all dimensions that compose the stored data block. For the example, in the schedule shown in Listing 5.1, the block size of ofmap at level 0 is calculated as $B_0 \times K_0 \times Y_0 \times X_0 = 1 \times 4 \times 16 \times 16$. Since at level 1, ofmap is split into 2×2 subblocks, one of them is stored at level 0, the ofmap block size at level 1 is $B_0 \times B_1 \times K_0 \times K_1 \times Y_0 \times Y_1 \times X_0 \times X_1 = 1 \times 4 \times 16 \times 2 \times 16 \times 2$. In summary, the ofmap block size at level i is the product of the tiled sizes of X, Y K and B $(\mathbf{O}[\prod_{j=0}^{i} B_j][\prod_{j=0}^{i} K_j][\prod_{j=0}^{i} Y_j][\prod_{j=0}^{i} X_j])$. Similarly, F_X, F_Y, C and K composes the weight block, thus we store weight block at level i to be $\mathbf{W}[\prod_{j=0}^{i} K_j][\prod_{j=0}^{i} C_j][F_Y][F_X]$. We can adopt the same approach to calculate the storage for ifmap at each level. Formally we can express the block size of each type as:

$$\mathrm{s}_i = \prod_{d \in D} \prod_{j=0}^{i} d_j \tag{5.1}$$

Here d_j, the loop bound at level j, is the number of subblocks in the dimension that composes the data block. If level 0 is register file, d_0 is the number of elements inside the register file in that dimension. For the example in Listing 5.1, the bounds of loop x and y at level 1 are both 2, thereby creating 4 tiles in total. Only the bounds of the dimensions that composes the data block (which belongs to D) are taken into account. The composed dimensions for ifmap, ofmap and weight are summarized in Table 5.1.

5.1.1 Performance Analytical Model

With the given hardware resource (compute units, memory sizes, maximum memory bandwidth) and input layer configurations, based on the roofline model [93, 99, 48], performance is determined by two factors — required bandwidth and computation resource utilization. Calculating the required bandwidth is easy: with data blocks assigned into memories, the amount of data required to be transferred between each adjacent levels is determined. Since we use a double buffer at each level, only the average bandwidth matters. Assume each data block is only evicted after being exhaustively reused by the computation within all inner loops, the average required bandwidth is the summation of the required bandwidth of each data block, which is estimated as the data block size divided by the number of cycles that compute on this data block. To summarize, required_bw$_i$ computes the required bandwidth of each data block at level i as:

$$\text{required\_bw}_i = \frac{s_i}{\text{cycles}_i} \tag{5.2}$$

In this equation, s_i is the data block size at level i, and cycles$_i$ is the number of cycles need to compute the results from this data block before replacing it with the next one. This cycle count is roughly the number of operations that reuse this data block divided by the actual number of operations processed in parallel. The later one is also affected by the bandwidths of the memories at level $i-1$ to level 0. With the dependency between cycles$_i$ and required_bw$_{i-1}$, to estimate total number of cycles cycles$_{L-1}$, we start from level 0 and recursively determine whether each level is bandwidth limited till the last level.

At level 0, the data is normally available in register file to be referenced by multiply-and-accumulate (MAC) units. Given a dataflow with constant unroll factors, the throughput T is proportional to the MAC units utilization rate. We can compute cycles$_0$ as the ratio between total number of operations to perform and parallelism it achieves, which is the product of all unrolled factors.

$$\begin{aligned} \text{cycles}_0 &= \frac{\#\text{required\_op}}{\prod_{uf \in UF} uf_i} \\ \text{comp\_utilization} &= \frac{\prod_{uf \in UF} uf_i}{\#\text{MAC}} \leq 1 \end{aligned} \tag{5.3}$$

Here UF is the set containing all unrolled loops, uf_i is the unrolled factor of each loop, #MAC is the total number of MAC units. When the total unrolled factor equals the total number of MAC units, computation resource is fully utilized, thus the peak performance can be reached at this level. If the total unrolled factor is smaller than the total number of MAC units, it suffer from load imbalance issue. We constrain the total unrolled factor to not exceed total number of MAC units to prevent utilization rates larger than 1. After obtaining cycles$_0$, together with s_0 we can calculate required_bw$_1$. By analyzing the provided memory bandwidth including memory port width and count, we can determine whether cycles$_0$ is limited by communication or computation, and use the larger number as cycles$_0$ to compute cycles$_1$. This process is recursively repeated till level $L-1$ to evaluate overall performance.

5.1.2 Energy Analytical Model

The total energy is composed of the compute and memory energies. The compute energy is the product of the number of operations and the energy consumed by each operation. The first one is only related to the input layer parameters, thus given the energy cost of a MAC unit, the total compute energy is a constant. The only energy consumption that is dependent of schedules are memory and communication energy. To quantify these energy costs in an L-level hierarchy, we adopt a model similar to [13], which computes the memory energy at each level as the product of the number of accesses to that level and per access energy cost. The total numbers of accesses are affected by data reuse RT_i at different memory levels of the three data types. Communication costs will be handled using a similar approach. This approach gives the total memory energy as:

$$E = \sum_{i=0}^{L-1} \#\mathrm{acc}_i \times e_i \quad \text{where } \#\mathrm{acc}_i = \prod_{j=i}^{L-1} \mathrm{RT}_j^{if} + \prod_{j=i}^{L-1} \mathrm{RT}_j^{of} + \prod_{j=i}^{L-1} \mathrm{RT}_j^{w} \tag{5.4}$$

Here e_i is the energy of accessing the ith level once. RT_i^{if}, RT_i^{of} and RT_i^{w} are the data reuse rates of ifmap, ofmap and weight respectively. **Data reuse RT_i is defined as the number of times the data are accessed by level $i-1$ during its lifetime at level i.** Take ifmap as an example, if ifmaps can not be fully buffered at level $i-1$, each time applying a new filter requires refetching blocks of ifmaps from level i to $i-1$. With the schedule in Listing 5.1, there are 8 filters at level 2, thus ifmaps are read 8 times from level 2, $\mathrm{RT}_2^{if} = 8$. At level 1, filters don't switch, ifmaps are simply streamed from the current level to level 0 with no reuse at this level, $\mathrm{RT}_1^{if} = 1$. At level 0, ifmaps are not only reused by different filters, but also by different window locations within a filter, thus iterating over loops k, f_x and f_y all causes refetching ifmap pixels from level 0 to MAC units, $\mathrm{RT}_0^{if} = 4 \times 3 \times 3$. If a register file is used at level 0, this indicates ifmap data is accessed by 36 times during its lifetime inside the register file. The total number of accesses $\#\mathrm{acc}_i$ is the summation of ifmap, weight and ofmap accesses. Given a schedule, based on the definition of data reuse, RT_i of all data types for buffer at level i can be quantified as the product of the bounds of all loops at level i that reuse the data block.

$$\mathrm{RT}_i = \begin{cases} \prod_{v \in V} v_i, & \text{for ifmap and weight} \\ 2\prod_{v \in V} v_i, & \text{for ofmap} \end{cases} \tag{5.5}$$

The ofmap reuse rate is doubled because each partial sum update requires one read and one write access. Here v_i, the loop bounds at level i, is the number of loop iterations that reuse the data blocks. In Listing 5.1, the bounds of loop k at level 2 is 8. Only the loops that reuses the data block (which belongs to V in Table 5.1) are considered. The loops that can reuse ifmap, ofmap and weight respectively are summarized in Table 5.1. To compute the reuse of ofmap at level i, we perform the product of the bounds of f_x, f_y, c loops, the reuse to ofmap buffer $\mathbf{I_i}$ is $F_X F_Y \prod_{j=i}^{L-1} c_j$. Note, for

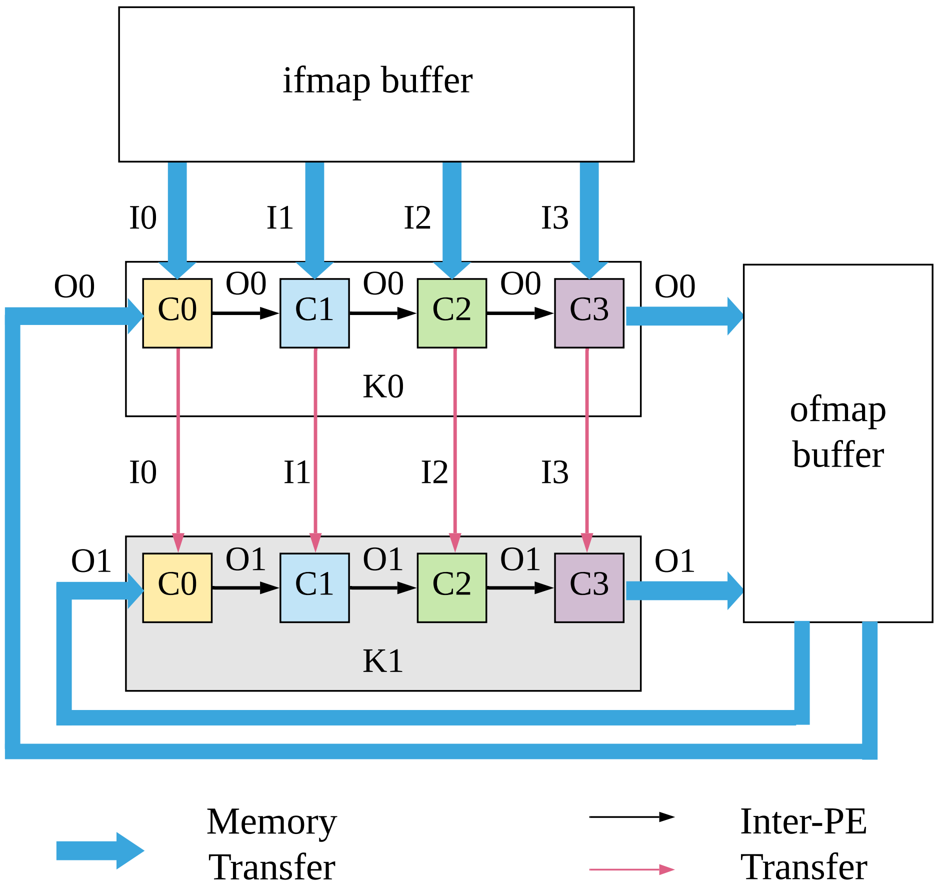


Figure 5.1: Data reuse with inter-PE communication. Ifmaps and Ofmaps are transferred once and 3 times on the inter-PE communication buses.

simplicity, we don't tile f_x f_y loops. In a similar way, we compute the reuse for the weight buffer $\mathbf{W_i}$ as $\prod_{j=i}^{L} x_j \prod_{j=i}^{L} y_j \prod_{j=i}^{L} b_j$.

We treat communication costs in a similar way. For the energy cost of communication we treat memories inside neighbor PEs as an additional level in the hierarchy. Figure 5.1 illustrates one example of the inter-PE communication of a systolic array. Two rows convolve with different filters to produce two output channels simultaneously, with unrolling loop k by 2. Since ifmaps are reused between the two rows, each ifmap pixel is first fetched from ifmap buffer, then transferred from the first row to the second one once via the inter-PE communication bus. Meanwhile, with unrolling loop c by 4, 4 columns of PEs reuse ofmaps. Every time to update an ofmap pixel in ofmap buffer, besides the first time and last time to read and write ofmap buffer, the rest $4 - 1 = 3$ times of data transfers are from the leftmost PE to the rightmost PE. Thus the total inter-PE communication can

Table 5.2: Energy per 16-bit access with various register file (RF) and SRAM sizes, and for a MAC operation, one hop communication cost and a DRAM access.

RF Size	Energy (pJ)
16 B	0.03
32 B	0.06
64 B	0.12
128 B	0.24
256 B	0.48
512 B	0.96
MAC	0.075
Hop	0.035

SRAM Size	Energy (pJ)
32 KB	6
64 KB	9
128 KB	13.5
256 KB	20.25
512 KB	30.375
DRAM	200

be determined by the number of accesses to the buffers shared by the PEs, and the PE counts that reuse the data. These two factors can be calculated based on the reuse rate and the unroll factors of corresponding loops. To compute the data reuse of the inter-PE communication level, we leverage the reuse rate RT_{i+1}:

$$\mathrm{RT}_i = \begin{cases} \mathrm{RT}_{i+1} \prod_{u \in V \cap U} u_i - 1, & \text{for ifmap and weight} \\ \frac{1}{2}\mathrm{RT}_{i+1} \prod_{u \in V \cap U} u_i - 1, & \text{for ofmap} \end{cases} \tag{5.6}$$

Here u_i is the unrolled factor of the loop that is unrolled and also reuses the data block (which belongs to V). Note, the product is subtracted by 1 in the end is because the first data transfer is from the shared buffer to the first PE, not on the inter-PE communication bus. Note, for the ofmap pixel, the inner-PE communication is calculated with respect to the update counts in the global buffer, thus we take $\frac{1}{2}\mathrm{RT}_{i+1}$ instead of RT_{i+1} for ofmap in Equation 5.6.

Cost Model: The energy cost e_i in Equation 5.4 can be obtained from the input cost model. We present out energy cost model in Table 5.2, which are employed to conduct experiments and generate results. Nevertheless, our energy modeling works with different technology processes, and it is easy to supply new cost models to study more advanced technologies. Also, many of our observations in Section 5.3 are technology-independent.

We use CACTI 6.5 [63] to model SRAM arrays and tune its parameters to match the synthesized memories instantiated from our 28 nm commercial memory library. For small arrays and register files (RFs), we use the Cadence XtensaProcessor Generator [4] to extract energy numbers based on our standard cell library. Table 5.2 shows the energy cost (pJ/ 16 bits) of accessing memories with different size. Since the wire length of transferring data from a memory is roughly the side length of the memory, which is about the square root of the memory area, and the memory area grows linearly with the memory capacity, we can interpolate Table 5.2 to obtain more energy cost numbers of memories with different sizes. Note that our energy ratios between memories and MAC are larger

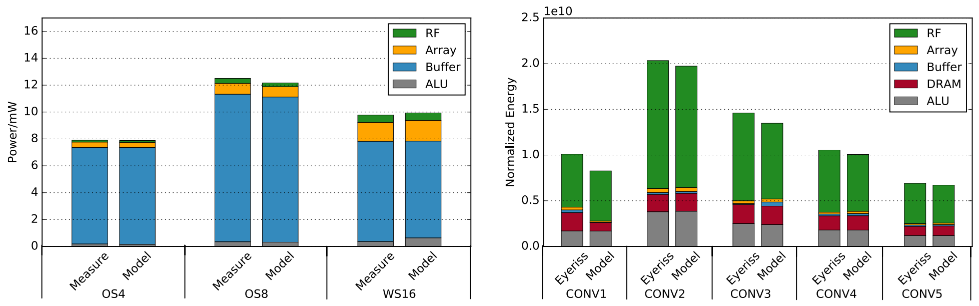


Figure 5.2: Analytical model validation. Left: energy breakdown comparison between actual synthesized designs and the analytical model. Right: energy breakdown comparison between reported Eyeriss model and our model.

Table 5.3: ASIC designs for model validation.

Name	Dataflow	PE Array	RF	SRAM
OS4	X	1D, 4	32 B	32 KB
OS8	X	1D, 8	64 B	64 KB
WS16	$C \mid K$	2D, 4×4	64 B	32 KB

than those reported in Eyeriss [13]. There are several reasons: we use a 28 nm technology instead of 65 nm; our memory is highly banked with higher energy cost; and our MAC units consume lower energy as their activity factors are relatively low with data stationary patterns.

For the inter-PE level, we use the hop energy in the table as the communication cost for fetching the data from any neighbor PE, in other words, cost for travelling one hop. This communication cost is estimated as the energy on the wires that connect the two neighbor PEs plus the energy consumed by the registers and muxes that are used for systolic data transfer. Ideally, the wire energy should be estimated as proportional to the wire length that is measured from the mid-point of one PE to the mid-point of the next connected PE. Using this estimation, with larger register file or more MACs allocated inside the PE, the wire length grows with the PE area size, so does the communication cost. However, for the design we explored, it is not a major factor, which we will demonstrate in Section 5.3. Hence, we use a average approximation of the communication cost per hop as presented in Table 5.2. Nevertheless, we do distinguish the cost for different communication distances (Figure 3.8) as an improvement over [13]. The communication cost is approximated as linearly growing with the travel distance (number of hops transferred).

5.1.3 Analytical Model Validation

The analytical model is validated in two different ways. First, we have thoroughly validated the accuracy of our model by comparing its results to complete designs generated by our synthesis toolchain. Our extended Halide compiler generates C++ code specialized for Catapult High-Level Synthesis, which is then compiled to RTL designs in Verilog. We synthesized the RTL designs in a 28 nm technology using Synopsys Design Compiler. Standard cells and memory models from commercial vendors are used for power, performance, and area analysis. We use 16-bit arithmetic for inference tasks throughout this paper. All of our ASIC designs achieve 400 MHz frequency with no timing violations. For power analysis, the appropriate switching activities are set on all the primary ports and propagated through the design using the design tools. Table 5.3 shows three example designs we have generated in ASIC platforms. Figure 5.2 shows the energy comparison between our analytic model and post-synthesis results. The resulting errors are less than 2%.

In addition to validating our analytical model against the synthesized designs, we further validated our analysis methodology by comparing the results against prior work. With using the same energy cost model, our framework is also able to reproduce the results from [13] with small differences, as demonstrated in Figure 5.2.

5.2 Optimization Flow

With these computationally simple models, this section introduces a systematical optimization framework we developed to study the design space.[1] It searches for the schedule/mapping which optimizes the data movement across the memory hierarchy to improve memory energy efficiency and resource utilization. While schedule is only one plane in the 3D design space (as the yellow plane shown in Figure 3.6 composed by loop blocking and dataflow axes), to create the optimal hardware acceleration solution, our optimizer additionally optimizes the hardware resource allocation as well. Furthermore, to obtain deeper understanding of the shape of the 3D design space introduced in Section 3.3, the optimizer investigates the impact of each factor independently using the loop-based taxonomy presented in Section 3.2.

The optimizer can be employed to realize four different goals:

1. Analyze the impact of resource allocation, loop blocking, dataflow individually with fixing other factors.
2. Given hardware architecture and resource allocation, search the optimal schedule/mapping for input layers that achieves highest resource utilization and throughput.

[1]The source is available at https://github.com/xuanyoya/CNN-blocking

3. Given hardware architecture and resource allocation, find the most energy efficient schedule/mapping while satisfying user specified requirements (for example, throughput requirement).

4. Given a set of DNN models, find the most efficient memory system configuration and the corresponding schedule.

5.2.1 Optimizer Structure

With the performance energy analytical model developed in Section 5.1.1 and Section 5.1.2, finding the optimal accelerator design now becomes an optimization problem of minimizing E or #cycle over the 3D design space. For instance, to determine the most energy efficient design, e_i is determined by the resource allocation (Table 5.2), and RT_i can be directly calculated from the dataflow and loop blocking schemes. Therefore, the most energy efficient design requires optimizing both resource allocation and schedule, so that it can minimize access cost e_i while meantime capturing as much data reuse RT_i as possible. To perform optimization for e_i and RT_i, we create an optimizer on top of the analytical models to explore the 3D space, as presented in Figure 5.3 and Figure 5.4.

The optimizer in Figure 5.3 combines a schedule generator and cost analysis engine, and achieves the first three goals. Built on top of this design, the optimizer in Figure 5.4 adds a memory system configuration generator to enumerate various hardware configurations, thereby enabling it to jointly optimize the hardware resource and schedules. Among the three modules, the schedule generator is used to iterate over all possible schedule candidates. The cost analysis engine uses the iterated schedule to quantify the overall data movement energy cost and/or throughput with a given memory hierarchy. Composing the schedule generator and cost analysis engine creates a schedule optimizer, which enables a search for the most energy efficient or highest throughput schedule given a CNN layer configuration and other architecture constraints. The memory configuration generator explores all the possible combinations of buffer parameters, and calls schedule optimizer to perform cost estimation for each candidate. Adding the memory configuration generator on top of the schedule optimizer allows for optimizing the memory system along with the schedule.

The basic optimizer takes a cost model, configurations of the given architecture and layers as input, and conducts exhaustive search over the huge design space. More details about the input will be provided in Section 5.2.2. Also in later sections, we will present how we leverage the understanding of the space and insights about each factor to prune the large space for faster exploration.

Schedule Generator: To comprehensively explore the space, schedule generator exhaustively enumerates all combinations of loop blocking sizes, loop orders, loop unrolling choices. To achieve this exhaustive enumeration for a L-level hierarchy, the generator generates one schedule candidate at each iteration. The candidate is generated by recursively tiling each dimension $L-1$ times to produce d_{l-1}, d_{l-2}, ... d_0, with $\prod_{i=0}^{L-1} d_i = D$, where $(d, D) \in \{(f_x, F_X), (f_y, F_Y), (x, X), (y, Y), (c, C), (k, K), (b, B)\}$. Similar to example shown in Algorithm 2, d_{l-1}, d_{l-2}, ... d_0 are assigned from level $L-1$

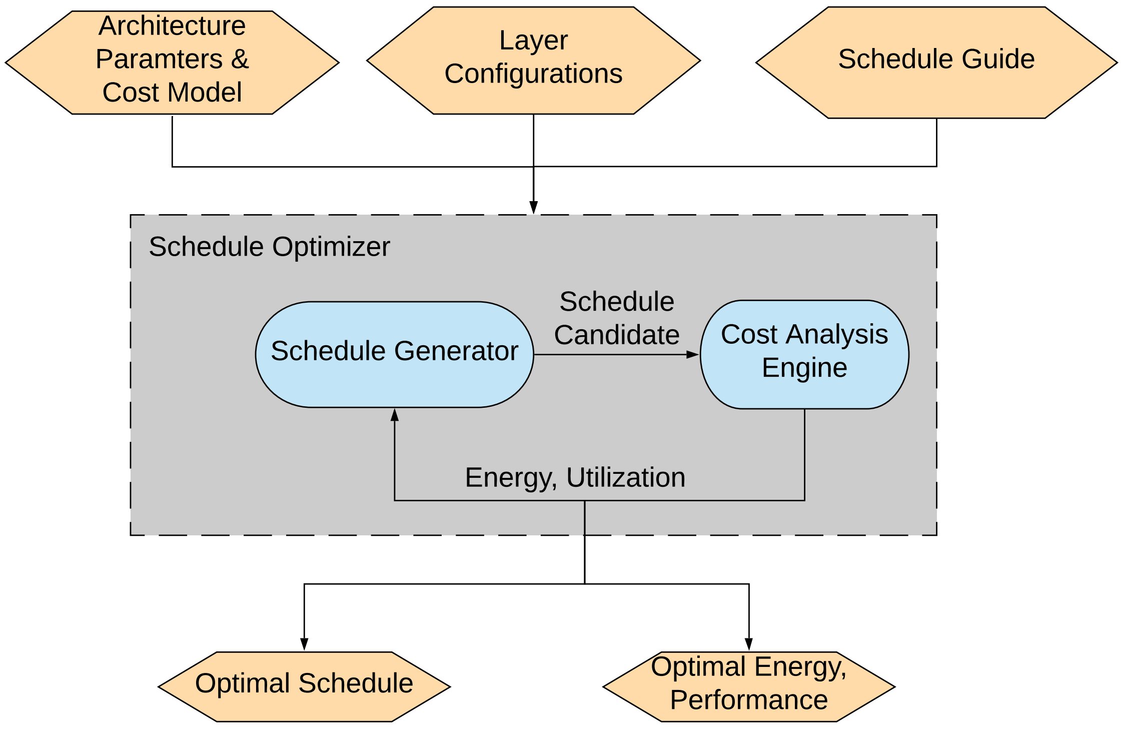


Figure 5.3: The schedule optimizer flow. The optimizer reports optimal schedule with its energy and performance. Orange hexagons are inputs and outputs of the framework. Schedule Generator generates schedule candidates and send to analysis engine for energy and performance evaluation.

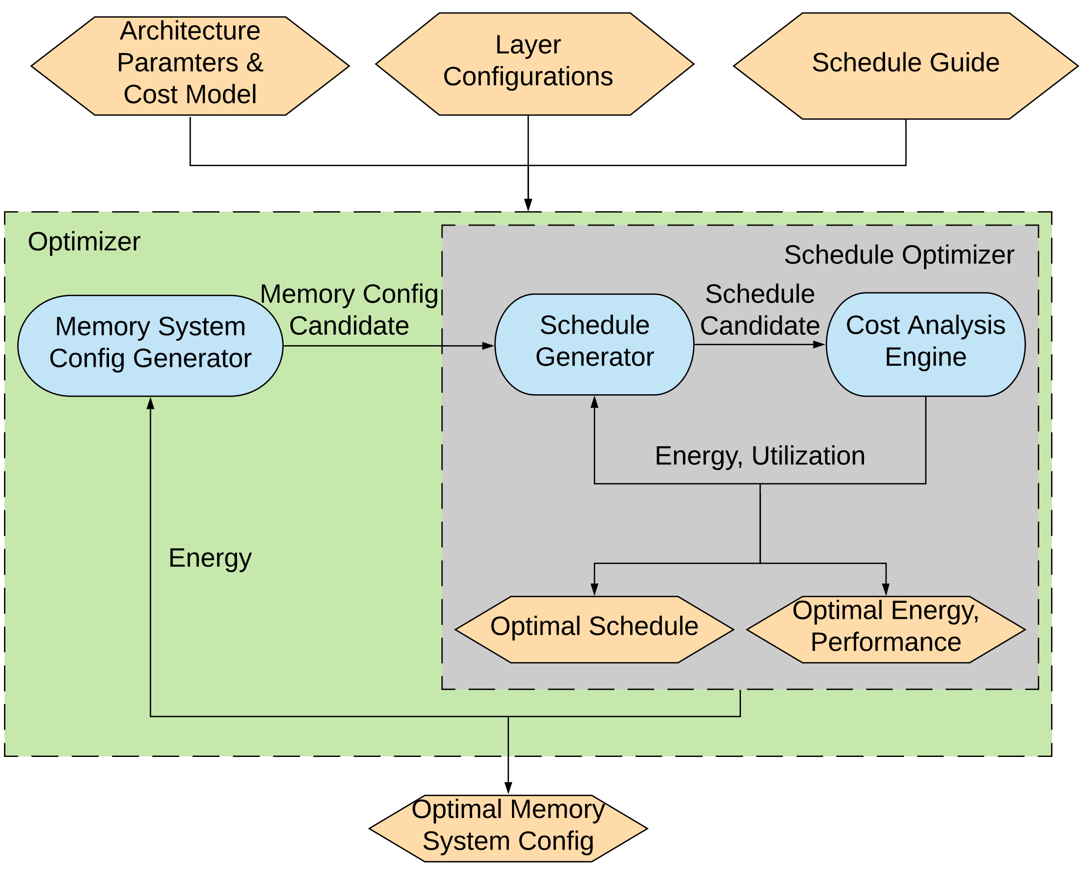


Figure 5.4: The optimizer flow. The optimizer jointly optimizes memory system and schedule, and reports the optimal design with the achieved energy and performance. Orange hexagons are inputs and outputs of the framework. Memory Configuration Generator generates configuration candidates and send to schedule optimizer for energy evaluation.

to level 0. After loop tiling, the generator enumerates all possible loop orders within each level to accomplish producing one schedule candidate. If the architecture has multiple parallel compute tiles and/or PEs, the generator also iterate over all possible loop unrolling choices. Specifically, after obtaining tiled loops nests, the generator chooses a subset loops to unroll from the ones at the level that has multiple parallel units. The unrolled factor for each selected is enumerated from 1 to D_i (the tiled size at that level).

However, such generated candidate may not always be valid. To reduce the runtime, the generator also checks each schedule to filter out invalid ones, before sending to the cost analysis engine for evaluation. The valid schedule must satisfy certain conditions, for example, the total required buffer size s_i by the schedule doesn't exceed the given allocated buffer size at each level, the total unroll factor is not larger than the number of physical parallel units at each level, and so on.

Memory System Configuration Generator: Since resource allocation affects access energy cost e_i, memory configuration optimizer explores the resource allocation choices to find the optimal memory system configuration. As shown in Figure 5.4, the optimizer is built by creating an memory configuration generator to wrap around the schedule optimizer. The memory configuration generator iterates overall all possible combinations of memory system configurations, and calls schedule optimizer to search for the most energy efficient schedule for the candidate memory system. Optimizing for multiple layers or a set of DNN models takes a similar approach as for single layer. To achieve the lowest energy for all the input layers, for each enumerated system candidate, it estimates the overall best energy efficiency by calling schedule optimizer to report the optimal energy for each layer.

5.2.2 Optimizer Input and Output

Input: The input to the optimizer consists of architecture parameters, a cost model for the hardware, layer configurations, and optionally a schedule guide, as presented in Figure 5.3 and Figure 5.4. Shown in Listing 5.2 is one sample specification of architecture parameters, and a cost model as inputs to the optimization framework. The architecture can be specified to contain arbitrary number of memory levels in the hierarchy. This example specifies 3 memory levels — DRAM, global buffer and RF per PE, the hierarchy from level 3 to level 0 are configured as the three levels of memory hierarchy (DRAM, global buffer, RF per PE), plus the array communication (inter-PE communication). Note, even without explicit buffer allocation, array (neighbor PEs) is treated as an additional level in the hierarchy, so that we can support direct inter-PE communication in systolic arrays. Additionally, Listing 5.2 provides the energy cost per access (pJ/16 bits) at each of the three levels in the memory hierarchy, which is generated by indexing to Table 5.2 with the corresponding buffer sizes. It also supplies the communication energy cost. The `BankSize` in the Listing is the bank size used for the memory at each level, which determines the number of ports, thereby the overall bandwidth of the memory at each level.

```
MemoryLevels: 3                           # number of memory levels
MemorySize: [512, 131072, Inf]            # size of [RF, global buffer, DRAM]
BankSize: [512, 8192, Inf]                # bank size of each memory (optional)
MemoryEnergyCost: [0.96, 20, 200]         # energy per access to [RF, global buffer, DRAM]
ArrayEnergyCost: [0.035]                  # energy for traveling one hop
SpatialSize: [[16, 16], 1, 1]             # 16 x 16 2D array
CommunicationMode: [broadcast, None, None] # broadcast bus
```

Listing 5.2: An example input specification for architecture parameters and cost mode. The indication of each field can be found in Section 3.3

The optimizer can take multiple layer configurations as input. When providing the configurations of all the layers in a network, the optimizer searches the memory system that achieves the highest energy efficiency for the entire given network. Each layer configuration includes fmap sizes, channel numbers, filter sizes, precision, etc.. The configuration of the first layer of AlexNet is illustrated in Listing 5.3. As mentioned in Section 2.1, a CONV layer processes the ifmaps of `input_fmap_channel` channels to produce ofmaps of `output_fmap_channel` channels, each ofmap channel of size `fmap_width` $\times$ `fmap_height`. The computation are performed in a batch size `batch_size`, with stride size `stride_width` and `stride_height` at the x- and y- dimensions respectively. The format supports both CONV and FC layers.

```
fmap_width: 56,                 # X
fmap_height: 56,                # Y
input_fmap_channel: 3,          # C
output_fmap_channel: 96,        # K
window_width: 11,               # FX
window_height: 11,              # FY
batch_size: 16,                 # B
stride_width: 4,                # SX
stride_height: 4                # SY
```

Listing 5.3: An example layer configuration for the first layer of AlexNet

The schedule guide is used by designer to restrict the design space to a subset for shorter search time. For example, if the experiment is only conducted to study the impact of loop blocking, we can specify certain reasonable dataflow in schedule guide to fix the dataflow choice, and only search the corresponding plane that is orthogonal to the dataflow axis. For example, Listing 5.4 provides one example schedule guide that forces the dataflow to be $X \mid Y$, with unrolling factor 16 for both dimensions. Such spatial mapping are specified to occur at the RF level, as a result, it lays out a 16×16 output tile onto the PE array, with the RF inside each PE buffers one output pixel. This schedule guide is also useful to avoid searching over the non-optimal region, after obtaining insights about the design space, as we can focus the optimal region in the schedule guide.

```
Level0: {                  # memory level
  X: {                     # loop to schedule
    unroll: 16             # unroll factor
  }
  Y: {                     # loop to schedule
    unroll: 16             # unroll factor
  }
}
Level1: {                  # memory level
  ...
}
```

Listing 5.4: An example schedule guide for dataflow $X \mid Y$

Output: By taking the input specifications of the architecture parameters, cost model, layer configurations and schedule guide, the schedule optimizer in Figure 5.3 and the optimizer in Figure 5.4 outputs the optimal schedules. As depicted in Figure 5.3, by leveraging the cost analysis engine to evaluate the energy and resource utilization rate of valid schedules, the schedule optimizer records the schedule that consumes the lowest energy while still achieves reasonable utilization rate. After iterating over all the schedules, the schedule optimizer reports the optimal schedule with its energy and performance results with the given hardware configuration.

The outputs of the optimizer in Figure 5.4 are the optimal memory configuration, with the optimal schedule for that system. When providing multiple input layer configurations, the optimal memory configuration is optimized for all the layers. Similar to the schedule optimizer, exhaustively evaluating all memory configurations takes long running times evaluating many obviously bad configurations. Some configurations can be eliminated as they can never be optimal designs, for instance, the designs that have larger register files than global buffers. To make it more general, the optimal memory system configurations often follows certain rules or patterns, which can be leveraged to speedup the search process. In the next section, we will discuss more about the observations we obtained from the experiments that can be used to prune the resource allocation space.

5.3 Results

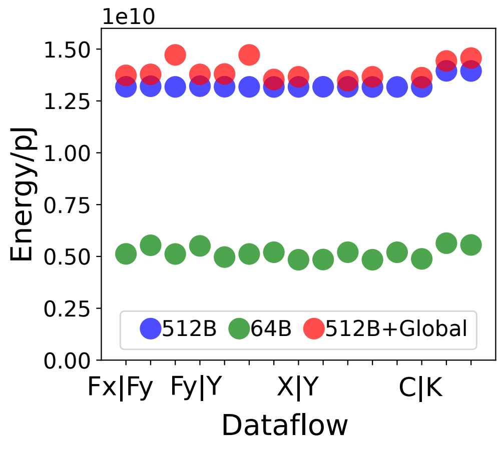
Using our dataflow taxonomy and the ability to rapidly generate and evaluate large numbers of accelerator designs with Halide, this section maps out the important features of each dimension of the design space. We begin by exploring different dataflow and loop blocking choices, and then consider hardware resource optimizations. Using the insights from these explorations, we introduce an efficient optimizer for DNN accelerators.

5.3.1 Impact of Dataflow and Loop Blocking

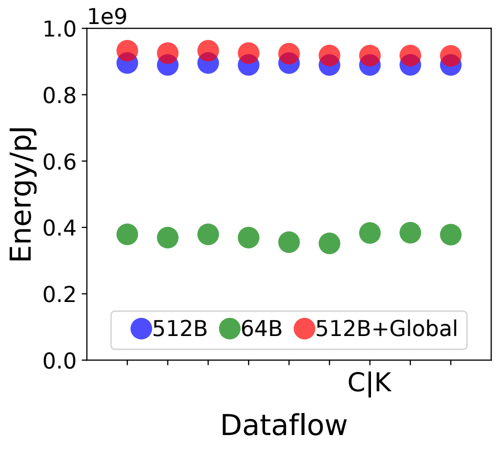
Figure 5.5 compares the energy efficiency of different dataflow choices. We use the CONV3 layer in AlexNet and 1×1 reduction layer 4C3R in GoogLeNet inception (4c) module as examples. The other layers have also been investigated, and share a similar trend. More DNNs will be studied in Section 5.3.2. For each dataflow, the loop blocking scheme is optimized to minimize the energy based on the analysis framework in Section 5.1, and the utilization ratio is constrained to be higher than 75%. The setting on the utilization ratio limits the performance degradation allowed. We have evaluated three different hardware configurations: the blue one is the same as Eyeriss [13], with 512 B register file (RF) per PE, 128 KB SRAM buffer, and 16×16 PE array; the red one uses a different array bus design, which disables inter-PE communication and makes all data broadcast from the global buffer; the green one uses a smaller, 64 B RF to lower its access energy (see Section 5.3.2). For the red configuration the communication cost is independent of the transfer distance, but slightly larger than the blue configuration. These points should amplify differences caused by communication. To lower the total energy required, the green configuration uses a smaller register file. From Figure 5.5, we can see that when optimized loop blocking schemes are applied, different dataflows all achieve similar and close-to-optimal energy efficiency on the same hardware configuration. Figure 5.5(b) and 5.5(d) also show the cases with batch size 1, which is most commonly used in mobile systems; the conclusion is consistent across different batch sizes.

This result is not sensitive to memory/communication model used. It remains true for a variety of scenarios, including using different layers, different NNs, different spatial array organizations, different PE array sizes, different models for communication cost estimation, and different memory configurations. For example, rather than building a 2-D PE array, we created a 256 PE 1-D systolic array. This design was only up to 0.4% worse than the 2-D array.

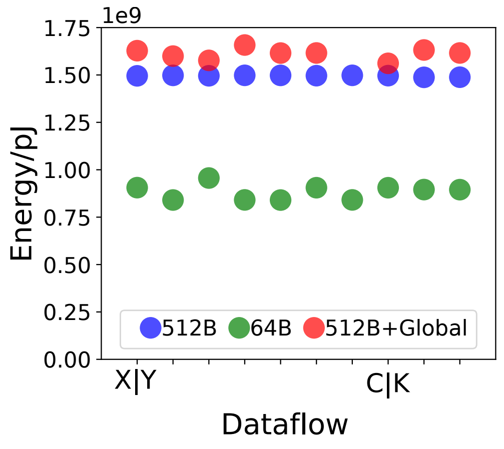
On the other hand, in Figure 5.6 we observe that the PE array utilization, and therefore the computation throughput, is slightly more sensitive to different dataflow choices than the energy efficiency for some convolution layers. Without replication (Figure 5.6(a)), the overall utilization can vary significantly and stay low for many dataflow choices. However, using proper replication substantially improves the utilization and eliminates most of the differences among all dataflows (Figure 5.6(b) and 5.6(c)). These results also imply, accelerators that can support a diversity of replication schemes using flexible communication such as CGRAs, Eyeriss V2 [14], will generally achieve higher overall utilization, compared to the ones with fixed interconnects. Here, the impact of the interconnect bandwidth to evaluate the overall performance is not included in the analysis, as prior works [14, 53] have already studied such impact. For the CONV3 layer in AlexNet, the $C\,|\,K$ dataflow achieves 20% higher utilization than the others such as $F_Y\,|\,Y$. This is because the channel dimensions C and K are typically the largest in most CONV and FC layers, so it is easier to unroll them onto a fixed-sized PE array with better load balance. From a performance perspective, we select the $C\,|\,K$ dataflow in the rest of this paper.



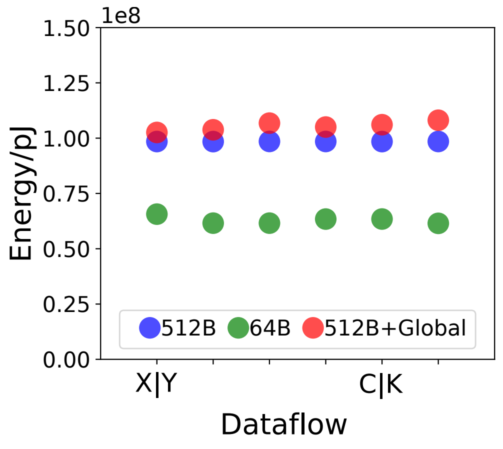
(a) Batch 16 (AlexNet).



(b) Batch 1 (AlexNet).

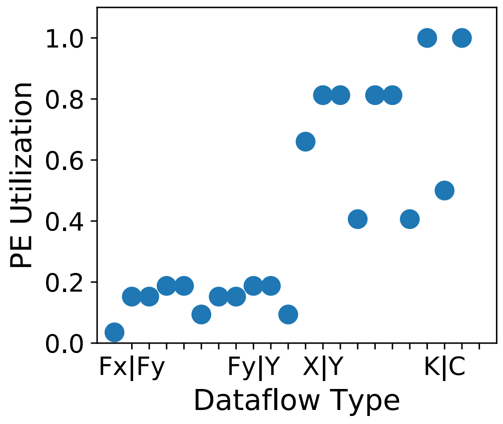


(c) Batch 16 (GoogleNet).

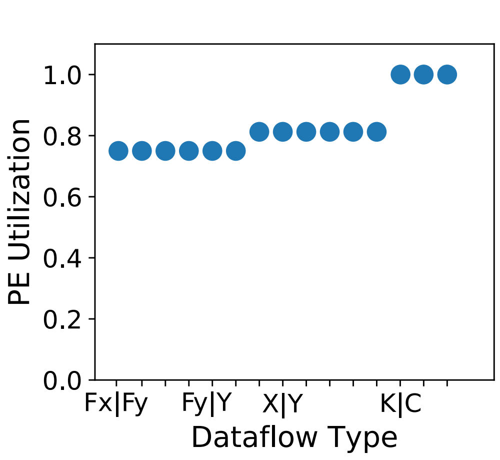


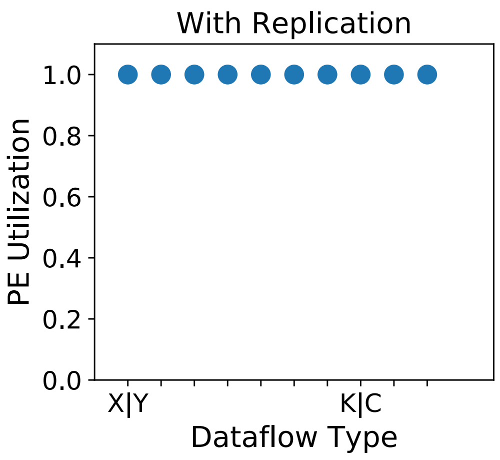
(d) Batch 1 (GoogleNet).

Figure 5.5: Design space of dataflow for AlexNet CONV3 and GoogLeNet 4C3R layers. Y-asix is the energy consumed to execute the entire batch. Different dataflows are shown horizontally, with only the most common choices labeled for clarity. All dataflows use replication and the optimal loop blocking schemes. Different colors represent different hardware resource allocations, and the energy cost is to complete the batch.



(a) No Replication (AlexNet)





(b) With Replication (AlexNet)

(c) With Replication (GoogleNet)

Figure 5.6: PE array utilization for the energy-optimal dataflow choices on AlexNet CONV3 layer with and without replication, and GoogleNet 4C3R layer with replication.

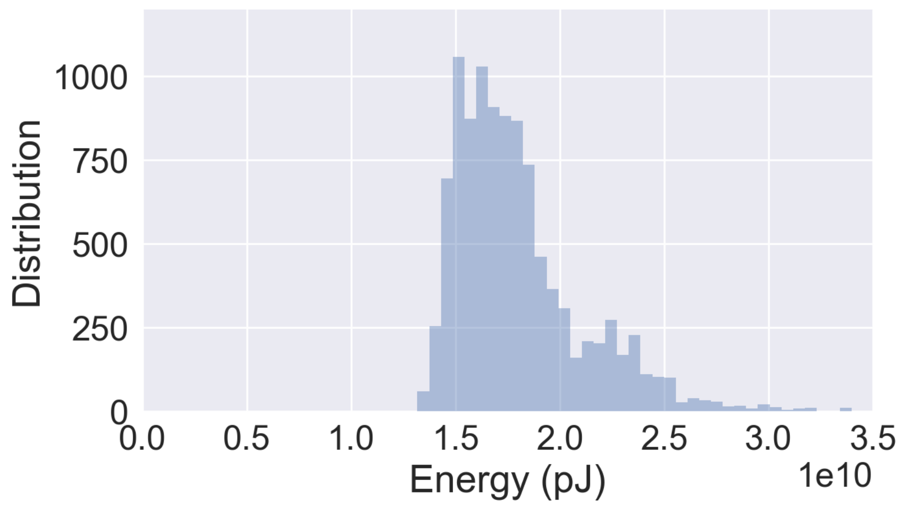


Figure 5.7: Design space of loop blocking for AlexNet CONV3 using dataflow $C\,|\,K$ with 512 B RF per PE.

Not all the computational layers have as much data sharing to exploit. Weight sharing in FC connected layers only comes from batching, and some applications limit the batch size to be small, even one. Interestingly, even in these computations the dataflow does not have a large influence on performance or energy. For computations with limited reuse, the data must come from the off-chip DRAM, or the last level on-die storage, if it is large enough. The storage properties at this level will limit the device's energy and performance, so for this class of application, the design of the computation units is less important.

Instead of dataflow choices, Figure 5.7 shows the design space of loop blocking for AlexNet CONV3, using a 512 B RF, corresponding to the blue configuration in Figure 5.5(a). We draw the energy distribution of all blocking choices as a histogram, with y-axis being the counts of the blocking choices that consume the corresponding energy on the x-axis. The energy variance of different blocking schemes is much more significant than that of dataflow, only 30% of the schemes fall within 1.25× of the minimum energy, with the distribution having a long tail on the higher energy side. Besides, the optimal loop blocking choices vary depending on the layer shapes. This indicates that loop blocking has a large impact on energy efficiency.

Observation 1: *With the same hardware resource, different dataflows are all able to achieve similar and close-to-optimal energy efficiency, as long as proper loop blocking and replication are used.*

In hindsight, this result is not surprising. When the DNNs exhibit enormous data reuse opportunities, regardless of the dataflow used, as long as high data reuse is achieved through proper loop blocking schemes, the resulting energy efficiency should be good. When the reuse is limited, the

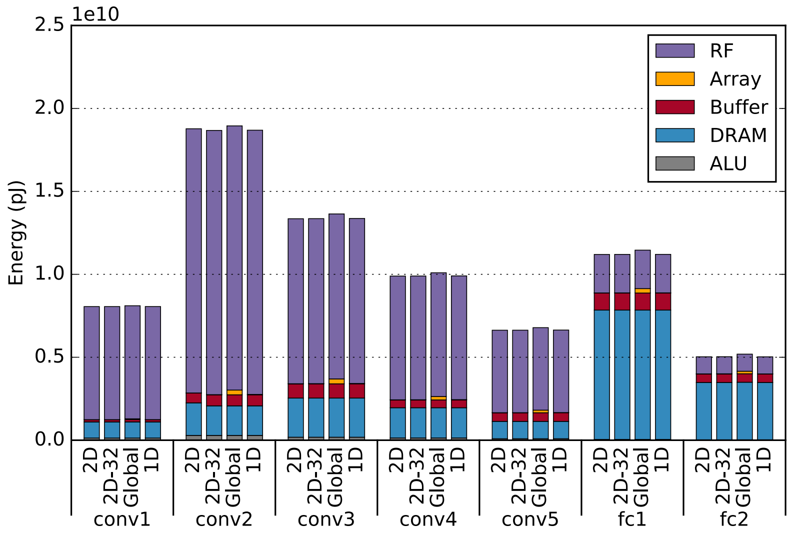


Figure 5.8: Energy breakdown of the optimal dataflows with different hardware configurations. 2D and global refer to the best blue and red points in Figure 5.5. 2D-32 and 1D change the PE array to 32×32, and 1D with 256 PEs respectively.

performance is limited by the bandwidth of the last level in the memory hierarchy instead of the PE array. These two situations are further illustrated in Figure 5.8, where the left bars with 512 B RF show the energy breakdown of the optimal dataflow for the blue configuration in Figure 5.5(a). For CONV layers with high reuse, most energy is consumed in the RF level rather than the array buses or intermediate buffers. By optimally blocking the computation, nearly all accesses (98%) occur at the RF level, making it the dominant energy component. For FC layers with limited reuse, most of the DRAM energy is inevitable, since the data have to be fetched at least once from off-chip (compulsory misses). On the other hand, the on-chip communication is generally only a small portion of the total energy, and therefore different dataflows do not substantially impact the overall energy efficiency. However, to support various replication schemes for better computation resource utilization requires flexible communication between the PEs.

5.3.2 Impact of Hardware Resource Allocation

Another interesting result from Figure 5.8 is that the total energy is always dominated by the RF level with a 512 B RF. This suggests that resource allocation may be suboptimal. Figure 5.9 shows the impact of memory resource allocation on energy efficiency. The energy is accumulated across all layers (including FC layers) in AlexNet, and contains both computation and memory access portions. It indicates that using a smaller RF size such as 32 or 64 B can significantly improve the total energy efficiency by up to 2.6×. If we also increase the global SRAM buffer size, the energy

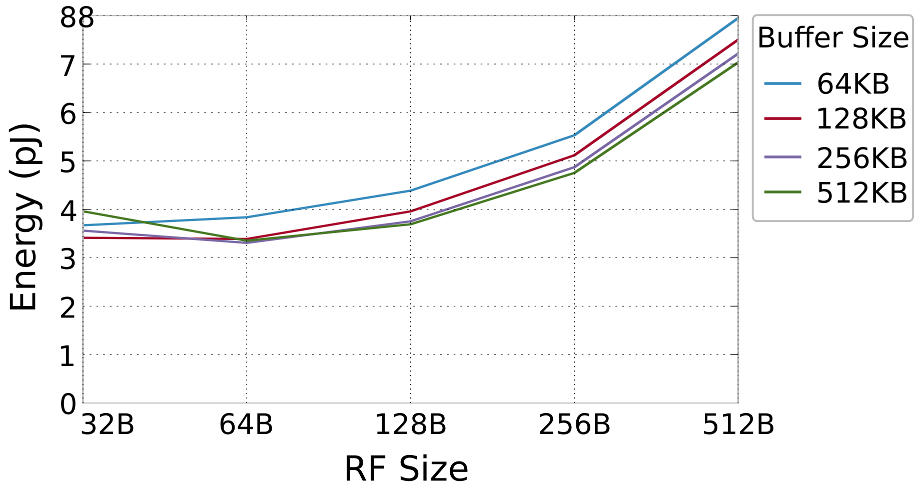


Figure 5.9: Memory hierarchy exploration with dataflow $C\,|\,K$. Different RF sizes per PE are shown horizontally. Lines with different colors correspond to different SRAM buffer sizes.

efficiency can further improve. However, when SRAM buffer size grows beyond 256 KB, the benefit becomes negligible. Given the significant area cost, it is not always necessary to use large global buffers.

We further look at the energy breakdown of using a 64 B register file, shown in Figure 5.10. Compared with a 512 B register file, the right bars in Figure 5.10, which give the energy breakdown of using a 64 B RF, illustrate that the energy decreases dramatically for all the CONV layers due to the much lower energy cost per access of the smaller RF. At the same time, more accesses go to the inter-PE array level and the global buffer, since the smaller RF captures less data reuse inside each PE. This also explains why dataflows with 64B RF (green dots in Figure 5.5) has a large variance. But reducing the RF size has almost no impact on the DRAM energy, as the data are still efficiently reused in the global buffer. Overall, a smaller RF achieves significantly better energy efficiency, with a more balanced energy breakdown among different memory hierarchy levels.

Observation 2: *The total energy of an efficient system should not be dominated by any individual level in the memory hierarchy.*

Observation 2 also explains why some existing output-stationary and weight-stationary designs do not perform well, as discussed by [13]. Those designs cannot capture sufficient reuse at the RF level, and result in high energy consumption at the DRAM level, which dominates the overall energy.

However, there is an exception for Observation 2. When DRAM dominates the total energy but the number of DRAM accesses is already minimized (fetching the input once and writing back output once), the DNN is memory bound, and based on Amdahl's law, little further optimization can be achieved for the memory hierarchy. This is the case particularly for a batch size of 1, and MLPs and LSTMs that contain many FC layers.

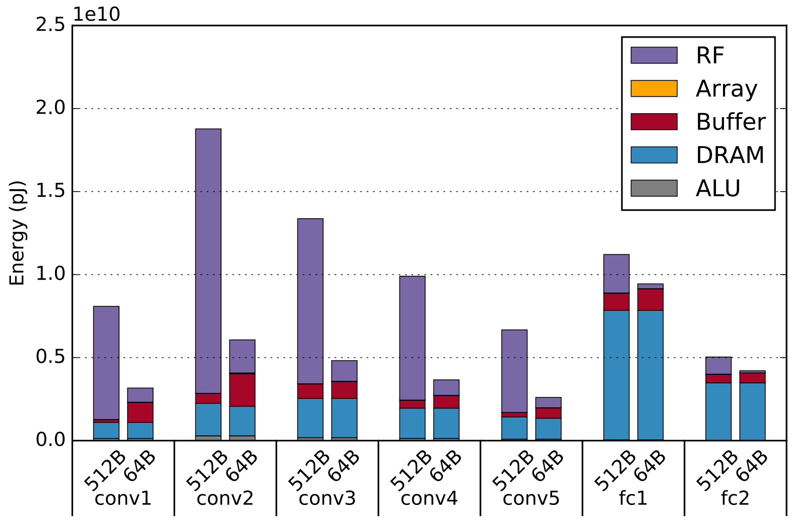


Figure 5.10: Energy breakdown comparison between 512 B and 64 B RF sizes with the same dataflow. Using a 64 B RF reduces the overall energy significantly.

By rearranging the memory sizes in the current hierarchy, we reap a significant efficiency improvement. We also investigate whether changing the hierarchy itself can further improve the energy efficiency. Due to the dominant role of the RF level, we add another level of private register file, and plot the resulting impact on energy in Figure 5.11. We again use the $C \mid K$ dataflow, but other dataflows have a similar trend.

We normalize the total energy against that of using one-level register file with the optimal size (64 B). The energy reduction for the CONV layers in the network, is more than 30%. This reduction leads to an overall efficiency improvement of approximately 25%, by choosing 16 B and 256 B to be the two level register file sizes, and 256 KB to be the global buffer size. The energy for the overall network only reduces by 25% is because the FC layers, which are included in the total energy, have its locality exploited in the original memory hierarchy, and almost all the data (input, weights, and output) are only accessed from the main memory once. As a result additional levels of memory hierarchy don't improve the efficiency of the FC layer. We can expect a slightly higher efficiency improvement for ResNet [39] or GoogLeNet [89], as they are mostly composed of CONV layers.

In Figure 5.11, the largest energy efficiency improvement is obtained when sizing each memory level is based on the rule that the ratio of the on-chip storage sizes between the adjacent levels should be around 4 to 16. In an optimally-sized memory hierarchy, each memory level should shield most of the references it receives from the next level in the hierarchy. Since the energy cost of an access grows slowly with size, this leads to large changes in memory size. The optimal size for the 2-level registers is 8 B/128 B, and 8 or 16B/256B, which both have min ratio of 16. Sixteen of these

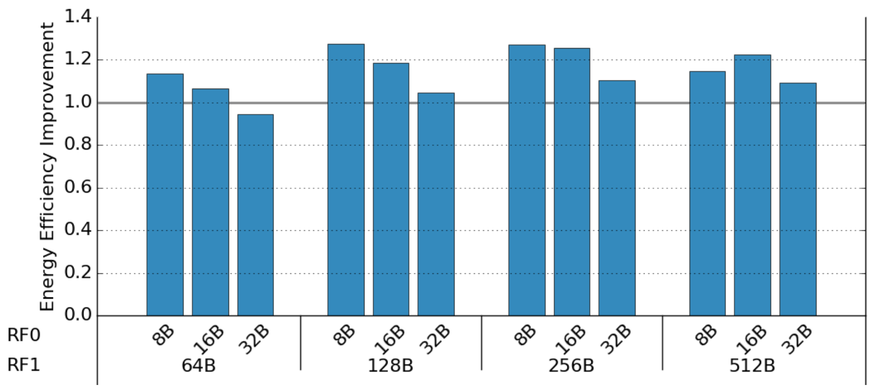


Figure 5.11: Overall energy efficiency improvement by adding another level of register file into the memory hierarchy. This improvement is calculated by dividing the overall energy of using two levels of RF by the optimal energy of using single level of RF. Bars that exceed 1.0 indicate energy improvements.

units create a 4 kB memory, which fetches data from a 256 kB or 512 kB buffer, which is a scale up of 64 to 128. The size of this buffer doesn't depend much on the configuration of the register hierarchy, and mainly depends on shielding most of the DRAM references. This is due to the fact that sizing RF sizes properly already captures more of the memory references at the RF level, the buffer only consumes a small portion of the total energy. Hence, even the buffer energy can be further optimized by adding additional memory levels, it will only make negligible impact on the overall energy efficiency at the cost of area overhead. Figure 5.9 with Figure 5.11, we find that the optimal global buffer sizes are both 512 KB, the same regardless of different numbers of hierarchy levels.

Figure 5.12 depicts the optimal memory resource allocation and the corresponding total energy for AlexNet when varying PE array size. We use only one level of RF here. These correspond to the optimal points on the optimizing plane shown in Figure 3.6. With increasing numbers of PEs, the optimal memory size at each level grows sub-linearly. Ideally we would like to keep the same amount of data reuse with constant storage capacity for each PE, which would lead to linearly increased memory size. However, the access cost of each memory level grows with its size (Table 5.2), which slows down the optimal capacity scaling to sub-linear. Between the RF and the SRAM buffer, the data reuse in RF is more critical. So the RF level has a stronger trend to keep constant capacity per PE. But it is eventually bound by the size of the next-level, i.e., the SRAM buffer.

Also we notice that the total energy reduces slightly with the increasing number of PEs. This indicates that larger arrays will have a significant effect on throughput and a small change in energy efficiency. The energy improvement is achieved by buffering more data on chip for reuse, and since most communication is nearest neighbor, the larger die does not increase communication costs significantly.

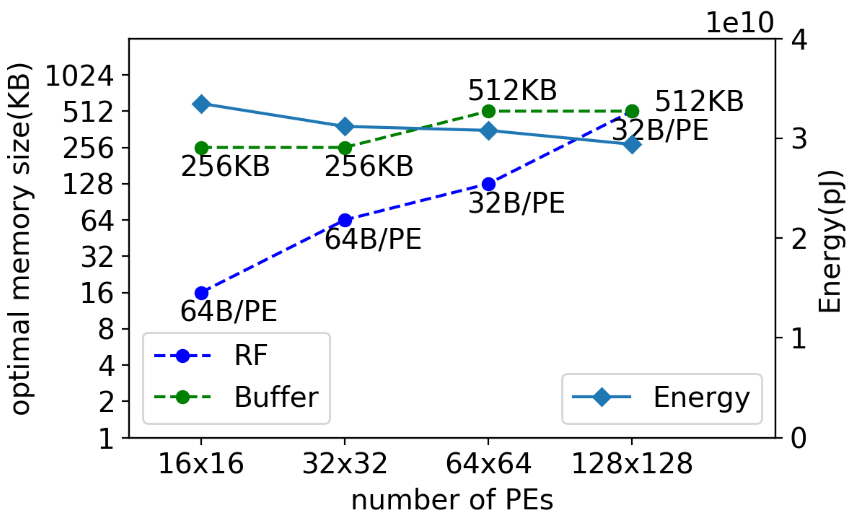


Figure 5.12: The optimal memory resource allocation and the corresponding total energy when varying PE array size.

5.4 An Efficient Optimizer

With the large number of hardware and software choices for DNN accelerators, exhaustive search for the optimal designs is usually infeasible. Instead, using the observations above, we can speed up the optimization process by pruning the search space and evaluating only a small number of candidates using the framework from Section 5.1.

We developed an auto-optimizer that efficiently finds energy efficient accelerator designs for given DNNs. The optimizer takes as input the DNN topology, the energy cost model, and various constraints such as the total chip area. First, according to **Observation 1**, we fix the dataflow to be $C \mid K$, and only search the design points on the optimizing plane in Figure 3.6. Next, we only evaluate a subset of hardware configurations with the optimal size of each memory level satisfying **Observation 2**, leveraging the rule that the ratio of the on-chip storage sizes and the adjacent levels should be around 4 to 16. It outputs an optimized design with corresponding Halide schedule primitives, which can then be fed into our hardware synthesis toolchain.

We use four CNNs, three LSTMs, and two MLPs as benchmarks to demonstrate the effectiveness of our efficient optimizer. All DNNs evaluated use 16-bit precision. The CNNs are AlexNet, VGG-16 [83], MobileNet [45], and GoogleNet [89] with batch size 16. The LSTM-M and LSTM-L are proposed by Google for sequence-to-sequence learning [88] with embedding sizes 500 and 1000. We also study the Recurrent Highway Network (RHN) [106]. The MLPs are from [17] with batch size 128. We use two baselines, both using dataflow $C \mid K$, which are the two left columns in Figure 5.13. The smaller chip uses a memory hierarchy similar to Eyeriss [13], and 16×16 PE array, whose area and power budgets are suitable for mobile platforms. The larger chip uses 128×128 PE array with

8 B register per PE, 64 KB for first-level global buffer, and a 28 MB second-level global buffer, similar to cloud-based accelerators such as TPU [48].

Figure 5.13 demonstrates the energy efficiency gain achieved by the efficient optimizer. We can improve the energy efficiency by up to 3.5×, 2.7×, and 4.2× for VGG-16, GoogleNet and MobilelNet, up to 1.6× for LSTMs, and up to 1.8× for MLPs. The optimal memory hierarchy uses 16 B and 128 B for the first-level and second-level register files, with a 256 KB global SRAM double buffer. This hardware configuration is shared by all the layers in the DNNs. Different from Eyeriss, the overall system energy consumption is not dominated by the RF level. The energy efficiency for the nine benchmarks are 1.85, 1.42, 0.87, 0.35, 0.49, 0.47, 0.5, 0.46, and 0.48 TOPs/W, respectively. Notice that even though the larger system has a smaller RF size, its energy is better than the smaller system. This is because with a much larger global SRAM buffer, it can store all the input and output data and the layer weights, and the accesses to DRAM are eliminated when switching to the next layer.

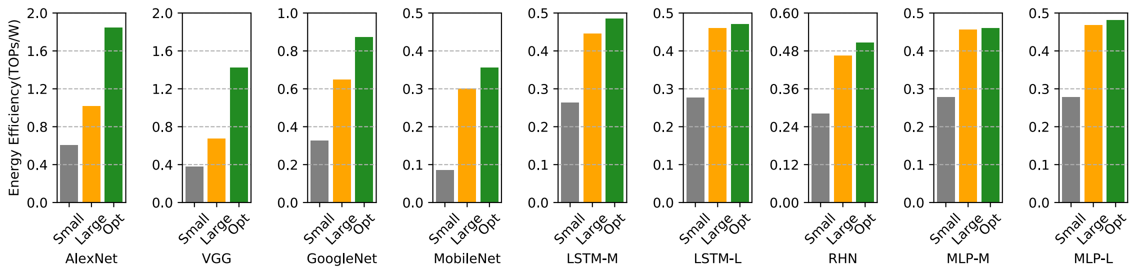


Figure 5.13: Overall energy efficiency improvement by using the auto-optimizer.

In summary, using the systematical analysis framework to study the global design space, we observe many dataflows can achieve similar and near-optimal energy efficiency, since properly blocking the computation can capture most memory references at the memory level closest to the computation units. The parameters in the space that make significant impact on efficiency are the loop blocking choice and the memory hierarchy design. The results indicate making register files larger does not always make them better, especially when the energy cost of the operations is small, and confirms the power of deep memory hierarchies.

Chapter 6

Conclusion

DNN applications are becoming pervasive, due to the superior accuracy on many modern intelligence problems, including recognition, detection, sequence modeling and so on. These workload are extremely computationally intensive, and require significant memory. Despite of the massive dataset involved in the computation, the algorithms have high data locality and parallelism, which previous researcher have exploited in a variety of DNN accelerators.

Even after many papers describing efficient DNN accelerators that have been published, there was little clarity about which hardware or software parameters were critical for efficiency. To help address this issue, we showed that the design space of DNN accelerators can be viewed as a 3D design space for DNNs, where one axis is loop blocking, another is the hardware dataflow used, and the last is the hardware resource allocation. More importantly, we realized that these three dimensions can be precisely and concisely described as the schedules of loop transformations. Thus we transform the problem of describing the design space of DNN accelerators into the problem of blocking and scheduling the (upto) seven level loop nest of a DNN.

Since Halide scheduling language already provides the required facilities for describing loop transformations for software programs, it provided a great starting point for our hardware generation framework. With only small extensions, we were able to express both the micro-architectures and dataflow mappings for existing DNN accelerators as schedules of a Halide program. Thus by extending the scheduling language and creating a hardware backend for Halide system, we were able to generate and fairly compare all proposed dense DNN accelerators.

Perhaps unsurprisingly given the large diversity of the existing DNN accelerators, the high locality in DNN algorithms means that there are many ways of partitioning the problem that achieve high efficiency. As long as the data reuse and the resource utilization are maximized by proper loop blocking and replication, hardware dataflow decisions aren't critical, and the results will be near optimal. The choices of loop blocking and the design of memory hierarchy have a larger effect on system efficiency, than the hardware dataflow. Again unsurprising in hindsight, optimal solutions try to balance energy among the different levels in the memory hierarchy. The large number of accesses to the memory next to the compute units really drives making these register files small, significantly smaller than we expected when we started. When designing future accelerators, we argue that deep memory hierarchies with properly sized memory at each level is important for DNNs, just as they are for CPUs. And the memory should be sized to accommodate efficient loop blocking that ensure any single memory level doesn't dominate the overall energy.

While this framework enabled us to better understand the DNN design space, there are a number of ways it could by further improved. The first one is a question of efficiency. Our current method of optimizing the memory resources and schedule enumerates through different memory options. It should be possible to speed this exploration by using a smarter joint search algorithm. Another limitation of our system is that it doesn't consider data resolution. In the last decade, outstanding breakthroughs have been made that successfully reduced the network size and memory footprint with negligible accuracy loss by optimizing the data precision. Combing the previously proposed quantization technique with the energy modeling in our optimization framework, we can replace the size of memory footprint with the energy cost as the optimization objective to directly optimize the algorithms for minimal energy cost. Furthermore, adding this extension can allow us to explore the opportunities of using different precision at each memory level with re-optimizing the schedule choices to achieve even better efficiency.

Another major improvement would be to extend the work on auto-schedulers for Halide to work for hardware generation as well. Researchers have already created auto-schedulers on top of the Halide scheduling system to search the optimal schedule that maximizes the performance for CPU and GPU targets [61, 5]. We should be able to integrate the simple hardware performance and energy analytical models we use in our optimizer into the more general auto-scheduler and explore how well it works on both the DNNs that we have covered, as well as the problems that need acceleration. We hope that the strong connections between loop transformations and hardware designs we uncovered in this work will make designing accelerators easier in the future. And our findings will highlight the importance of deep memory hierarchies with proper memory sizes for future DNN accelerators.

Bibliography

[1] Arm ML Processor. https://developer.arm.com/products/processors/machine-learning/arm-ml-processor/.

[2] Intel ETANN. https://en.wikichip.org/wiki/intel/etann.

[3] NVDLA. http://nvdla.org/.

[4] Tensilica customizable processor IP. http://ip.cadence.com/ipportfolio/tensilica-ip.

[5] Andrew Adams, Karima Ma, Luke Anderson, Riyadh Baghdadi, Tzu-Mao Li, Michaël Gharbi, Benoit Steiner, Steven Johnson, Kayvon Fatahalian, Frédo Durand, and Jonathan Ragan-Kelley. Learning to optimize halide with tree search and random programs. *ACM Trans. Graph.*, 38(4):121:1–121:12, July 2019.

[6] Jorge Albericio, Patrick Judd, Tayler Hetherington, Tor Aamodt, Natalie Enright Jerger, and Andreas Moshovos. Cnvlutin: Ineffectual-neuron-free deep neural network computing. In *43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 1–13, 2016.

[7] Altera. Intel fpga sdk for OpenCL. https://www.altera.com/products/design-software/embedded-software-developers/opencl/overview.html.

[8] Manoj Alwani, Han Chen, Michael Ferdman, and Peter Milder. Fused-layer CNN accelerators. In *49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–12, 2016.

[9] Erfan Azarkhish, Davide Rossi, Igor Loi, and Luca Benini. Neurostream: Scalable and energy efficient deep learning with smart memory cubes. *IEEE Transactions on Parallel and Distributed Systems*, 29(2):420–434, 2017.

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

[11] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Haichen Shen, Eddie Q Yan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Tvm: end-to-end optimization stack for deep learning. *arXiv preprint arXiv:1802.04799*, pages 1–15, 2018.

[12] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. DianNao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. In *19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 269–284, 2014.

[13] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. In *43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 367–379, 2016.

[14] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. Eyeriss v2: A flexible and high-performance accelerator for emerging deep neural networks. *arXiv preprint arXiv:1807.07928*, 2018.

[15] Yu-Hsin Chen, Tushar Krishna, Joel Emer, and Vivienne Sze. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. In *IEEE International Solid-State Circuits Conference (ISSCC)*, pages 262–263, 2016.

[16] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, and Olivier Temam. DaDianNao: A machine-learning supercomputer. In *47th Annual ACM/IEEE International Symposium on Microarchitecture (MICRO)*, pages 609–622, 2014.

[17] Ping Chi, Shuangchen Li, Cong Xu, Tao Zhang, Jishen Zhao, Yongpan Liu, Yu Wang, and Yuan Xie. PRIME: A novel processing-in-memory architecture for neural network computation in ReRAM-based main memory. In *43rd International Symposium on Computer Architecture (ISCA)*, pages 27–39, 2016.

[18] Nitin Chugh, Vinay Vasista, Suresh Purini, and Uday Bondhugula. A dsl compiler for accelerating image processing pipelines on fpgas. In *2016 International Conference on Parallel Architecture and Compilation Techniques (PACT)*, pages 327–338. IEEE, 2016.

[19] Jason Cong, Yiping Fan, Guoling Han, Wei Jiang, and Zhiru Zhang. Platform-based behavior-level and system-level synthesis. In *2006 IEEE International SoC Conference*, pages 199–202. IEEE, 2006.

[20] Jason Cong and Jie Wang. Polysa: Polyhedral-based systolic array auto-compilation. In *Proceedings of the International Conference on Computer-Aided Design*, ICCAD '18, pages 117:1–117:8, New York, NY, USA, 2018. ACM.

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

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

[23] Philippe Coussy, Daniel D Gajski, Michael Meredith, and Andres Takach. An introduction to high-level synthesis. *IEEE Design & Test of Computers*, 26(4):8–17, 2009.

[24] Ruizhou Ding, Zeye Liu, Rongye Shi, Diana Marculescu, and R.D. (Shawn) Blanton. Lightnn: Filling the gap between conventional deep neural networks and binarized networks. In *Proceedings of the on Great Lakes Symposium on VLSI 2017*, GLSVLSI '17, pages 35–40, New York, NY, USA, 2017. ACM.

[25] Bruce Draper, Walid Najjar, Wim Bohm, Jeffrey Hammes, Bob Rinker, Charlie Ross, Monica Chawathe, and José Bins. Compiling and optimizing image processing algorithms for fpgas. In *Proceedings Fifth IEEE International Workshop on Computer Architectures for Machine Perception*, pages 222–231. IEEE, 2000.

[26] Zidong Du, Robert Fasthuber, Tianshi Chen, Paolo Ienne, Ling Li, Tao Luo, Xiaobing Feng, Yunji Chen, and Olivier Temam. ShiDianNao: Shifting vision processing closer to the sensor. In *42nd Annual International Symposium on Computer Architecture (ISCA)*, pages 92–104, 2015.

[27] Clément Farabet, Berin Martini, Benoit Corda, Polina Akselrod, Eugenio Culurciello, and Yann LeCun. Neuflow: A runtime reconfigurable dataflow processor for vision. In *2011 IEEE Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, pages 109–116, 2011.

[28] Chang Gao, Daniel Neil, Enea Ceolini, Shih-Chii Liu, and Tobi Delbruck. Deltarnn: A power-efficient recurrent neural network accelerator. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 21–30. ACM, 2018.

[29] Mingyu Gao, Jing Pu, Xuan Yang, Mark Horowitz, and Christos Kozyrakis. TETRIS: Scalable and efficient neural network acceleration with 3D memory. In *22nd ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017.

[30] Mingyu Gao, Xuan Yang, Jing Pu, Mark Horowitz, and Christos Kozyrakis. Tangram: Optimized coarse-grained dataflow for scalable nn accelerators. In *Proceedings of the Twenty-Fourth*

International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '19, pages 807–820, New York, NY, USA, 2019. ACM.

[31] Vinayak Gokhale, Jonghoon Jin, Aysegul Dundar, Berin Martini, and Eugenio Culurciello. A 240 G-ops/s mobile coprocessor for deep neural networks. In *2014 IEEE Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, pages 696–701, 2014.

[32] Mentor Graphics. Catapult High-Level Synthesis. https://www.mentor.com/hls-lp/catapult-high-level-synthesis/.

[33] Yijin Guan, Zhihang Yuan, Guangyu Sun, and Jason Cong. Fpga-based accelerator for long short-term memory recurrent neural networks. In *2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 629–634. IEEE, 2017.

[34] Zhi Guo, Betul Buyukkurt, Walid Najjar, and Kees Vissers. Optimized generation of data-path from c codes for fpgas. In *Proceedings of the conference on Design, Automation and Test in Europe-Volume 1*, pages 112–117. IEEE Computer Society, 2005.

[35] Zhi Guo, Walid Najjar, and Betul Buyukkurt. Efficient hardware code generation for fpgas. *ACM Transactions on Architecture and Code Optimization (TACO)*, 5(1):6, 2008.

[36] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. Deep learning with limited numerical precision. In *Proceedings of the 32Nd International Conference on International Conference on Machine Learning - Volume 37*, ICML'15, pages 1737–1746. JMLR.org, 2015.

[37] Philipp Gysel, Mohammad Motamedi, and Soheil Ghiasi. Hardware-oriented approximation of convolutional neural networks. *arXiv preprint arXiv:1604.03168*, 2016.

[38] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A. Horowitz, and William J. Dally. EIE: Efficient inference engine on compressed deep neural network. In *43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 243–254, 2016.

[39] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *arXiv preprint arXiv:1512.03385*, 2015.

[40] James Hegarty, John Brunhaver, Zachary DeVito, Jonathan Ragan-Kelley, Noy Cohen, Steven Bell, Artem Vasilyev, Mark Horowitz, and Pat Hanrahan. Darkroom: compiling high-level image processing code into hardware pipelines. *ACM Trans. Graph.*, 33(4):144–1, 2014.

[41] James Hegarty, Ross Daly, Zachary DeVito, Jonathan Ragan-Kelley, Mark Horowitz, and Pat Hanrahan. Rigel: Flexible multi-rate image processing hardware. *ACM Transactions on Graphics (TOG)*, 35(4):85, 2016.

[42] Geoffrey E Hinton, Simon Osindero, and Yee-Whye Teh. A fast learning algorithm for deep belief nets. *Neural computation*, 18(7):1527–1554, 2006.

[43] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

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

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

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

[47] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross B. Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. *CoRR*, abs/1408.5093, 2014.

[48] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA)*, Jun 2017.

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

[50] Duckhwan Kim, Jaeha Kung, Sek Chai, Sudhakar Yalamanchili, and Saibal Mukhopadhyay. Neurocube: A programmable digital neuromorphic architecture with high-density 3D memory. In *43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 380–392, 2016.

[51] Jong Hwan Ko, Burhan Mudassar, Taesik Na, and Saibal Mukhopadhyay. Design of an energy-efficient accelerator for training of convolutional neural networks using frequency-domain computation. In *Proceedings of the 54th Annual Design Automation Conference 2017*, DAC '17, pages 59:1–59:6, New York, NY, USA, 2017. ACM.

[52] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. ImageNet classification with deep convolutional neural networks. In *25th International Conference on Neural Information Processing Systems (NIPS)*, pages 1097–1105, 2012.

[53] Hyoukjun Kwon, Michael Pellauer, and Tushar Krishna. MAESTRO: an open-source infrastructure for modeling dataflows within deep learning accelerators. *CoRR*, abs/1805.02566, 2018.

[54] Hyoukjun Kwon, Ananda Samajdar, and Tushar Krishna. MAERI: Enabling flexible dataflow mapping over DNN accelerators via reconfigurable interconnects. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, pages 461–475, New York, NY, USA, 2018. ACM.

[55] Yi-Hsiang Lai, Yuze Chi, Yuwei Hu, Jie Wang, Cody Hao Yu, Yuan Zhou, Jason Cong, and Zhiru 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*, pages 242–251. ACM, 2019.

[56] Huimin Li, Xitian Fan, Li Jiao, Wei Cao, Xuegong Zhou, and Lingli Wang. A high performance FPGA-based accelerator for large-scale convolutional neural networks. In *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–9, Aug 2016.

[57] Wenyan Lu, Guihai Yan, Jiajun Li, Shijun Gong, Yinhe Han, and Xiaowei Li. FlexFlow: A flexible dataflow accelerator architecture for convolutional neural networks. In *23rd IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 553–564, 2017.

[58] Daisuke Miyashita, Edward H Lee, and Boris Murmann. Convolutional neural networks using logarithmic data representation. *arXiv preprint arXiv:1603.01025*, 2016.

[59] Bert Moons and Marian Verhelst. A 0.3–2.6 tops/w precision-scalable processor for real-time large-scale convnets. In *2016 IEEE Symposium on VLSI Circuits (VLSI-Circuits)*, pages 1–2. IEEE, 2016.

[60] Thierry Moreau, Tianqi Chen, Ziheng Jiang, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. VTA: an open hardware-software stack for deep learning. *CoRR*, abs/1807.04188, 2018.

[61] Ravi Teja Mullapudi, Andrew Adams, Dillon Sharlet, Jonathan Ragan-Kelley, and Kayvon Fatahalian. Automatically scheduling halide image processing pipelines. *ACM Transactions on Graphics (TOG)*, 35(4):83, 2016.

[62] Ravi Teja Mullapudi, Vinay Vasista, and Uday Bondhugula. Polymage: Automatic optimization for image processing pipelines. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 429–443, New York, NY, USA, 2015. ACM.

[63] Naveen Muralimanohar, Rajeev Balasubramonian, and Norm Jouppi. Optimizing NUCA organizations and wiring alternatives for large caches with CACTI 6.0. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 40, pages 3–14, Washington, DC, USA, 2007. IEEE Computer Society.

[64] Walid A Najjar, Wim Bohm, Bruce A Draper, Jeff Hammes, Robert Rinker, J Ross Beveridge, Monica Chawathe, and Charles Ross. High-level language abstraction for reconfigurable computing. *Computer*, 36(8):63–69, 2003.

[65] Eriko Nurvitadhi, David Sheffield, Jaewoong Sim, Asit Mishra, Ganesh Venkatesh, and Debbie Marr. Accelerating binarized neural networks: Comparison of fpga, cpu, gpu, and asic. In *2016 International Conference on Field-Programmable Technology (FPT)*, pages 77–84. IEEE, 2016.

[66] Eriko Nurvitadhi, Jaewoong Sim, David Sheffield, Asit Mishra, Srivatsan Krishnan, and Debbie Marr. Accelerating recurrent neural networks in analytics servers: Comparison of fpga, cpu, gpu, and asic. In *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–4. IEEE, 2016.

[67] Alexandros Papakonstantinou, Karthik Gururaj, John A Stratton, Deming Chen, Jason Cong, and Wen-Mei W Hwu. Fcuda: Enabling efficient compilation of cuda kernels onto fpgas. In *2009 IEEE 7th Symposium on Application Specific Processors*, pages 35–42. IEEE, 2009.

[68] Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Brucek Khailany, Joel Emer, Stephen W Keckler, and William J Dally. SCNN:

An accelerator for compressed-sparse convolutional neural networks. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pages 27–40. ACM, 2017.

[69] Ardavan Pedram, Robert A Van De Geijn, and Andreas Gerstlauer. Codesign tradeoffs for high-performance, low-power linear algebra architectures. *IEEE Transactions on Computers*, 61(12):1724–1736, 2012.

[70] Maurice Peemen, Arnaud AA Setio, Bart Mesman, and Henk Corporaal. Memory-centric accelerator design for convolutional neural networks. In *31st International Conference on Computer Design (ICCD)*, pages 13–19, 2013.

[71] Jing Pu. *Programming Heterogeneous Systems From an Image Processing Domain Specific Language*. PhD thesis, Stanford University, 2017.

[72] Jing Pu, Steven Bell, Xuan Yang, Jeff Setter, Stephen Richardson, Jonathan Ragan-Kelley, and Mark Horowitz. Programming heterogeneous systems from an image processing DSL. *ACM Trans. Archit. Code Optim.*, 14(3):26:1–26:25, August 2017.

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

[74] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 519–530, New York, NY, USA, 2013. ACM.

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

[76] Oliver Reiche, Moritz Schmid, Frank Hannig, Richard Membarth, and Jürgen Teich. Code generation from a domain-specific language for c-based hls of hardware accelerators. In *Proceedings of the 2014 International Conference on Hardware/Software Codesign and System Synthesis*, page 17. ACM, 2014.

[77] Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

[78] John S.Brunhaver. *Design and Optimization of a Stencil Engine*. PhD thesis, Stanford University, 2015.

[79] Fabian Schuiki, Michael Schaffner, Frank K Gürkaynak, and Luca Benini. A scalable near-memory architecture for training deep neural networks on large in-memory datasets. *IEEE Transactions on Computers*, 68(4):484–497, 2018.

[80] H. Sharma, J. Park, D. Mahajan, E. Amaro, J. K. Kim, C. Shao, A. Mishra, and H. Esmaeilzadeh. From high-level deep neural models to FPGAs. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–12, Oct 2016.

[81] Yongming Shen, Mechael Ferdman, and Peter Milder. Overcoming resource underutilization in spatial CNN accelerators. In *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–4, Aug 2016.

[82] Yongming Shen, Mechael Ferdman, and Peter Milder. Maximizing CNN accelerator efficiency through resource partitioning. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA)*, Jun 2017.

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

[84] Linghao Song, Jiachen Mao, Youwei Zhuo, Xuehai Qian, Hai Li, and Yiran Chen. Hypar: Towards hybrid parallelism for deep learning accelerator array. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 56–68. IEEE, 2019.

[85] Mingcong Song, Jiaqi Zhang, Huixiang Chen, and Tao Li. Towards efficient microarchitectural design for accelerating unsupervised GAN-based deep learning. In *High Performance Computer Architecture (HPCA), 2018 IEEE International Symposium on*, pages 66–77. IEEE, 2018.

[86] Nitish Srivastava, Hongbo Rong, Prithayan Barua, Guanyu Feng, Huanqi Cao, Zhiru Zhang, David Albonesi, Vivek Sarkar, Wenguang Chen, Paul Petersen, et al. T2s-tensor: Productively generating high-performance spatial hardware for dense tensor computations.

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

[88] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to sequence learning with neural networks. *CoRR*, abs/1409.3215, 2014.

[89] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. *arXiv preprint arXiv:1409.4842*, 2014.

[90] Maxeler Acceleration Technology. MaxCompiler White Paper. https://www.maxeler.com/media/documents/MaxelerWhitePaperMaxCompiler.pdf.

[91] Kodai Ueyoshi, Kota Ando, Kazutoshi Hirose, Shinya Takamaeda-Yamazaki, Junichiro Kadomoto, Tomoki Miyata, Mototsugu Hamada, Tadahiro Kuroda, and Masato Motomura. Quest: A 7.49 tops multi-purpose log-quantized dnn inference engine stacked on 96mb 3d sram using inductive-coupling technology in 40nm cmos. In *2018 IEEE International Solid-State Circuits Conference-(ISSCC)*, pages 216–218. IEEE, 2018.

[92] Xuechao Wei, Cody Hao Yu, Peng Zhang, Youxiang Chen, Yuxin Wang, Han Hu, Yun Liang, and Jason Cong. Automated systolic array architecture synthesis for high throughput CNN inference on FPGAs. In *Proceedings of the 54th Annual Design Automation Conference 2017*, DAC '17, pages 29:1–29:6, New York, NY, USA, 2017. ACM.

[93] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: An insightful visual performance model for floating-point programs and multicore architectures. Technical report, Lawrence Berkeley National Lab.(LBNL), Berkeley, CA (United States), 2009.

[94] Xilinx. Vivado High-Level Synthesis: Accelerates IP Creation by Enabling C, C++ and System C Specifications. https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html.

[95] Xuan Yang, Jing Pu, Blaine Burton Rister, Nikhil Bhagdikar, Stephen Richardson, Shahar Kvatinsky, Jonathan Ragan-Kelley, Ardavan Pedram, and Mark Horowitz. A systematic approach to blocking convolutional neural networks. *arXiv preprint arXiv:1606.04209*, 2016.

[96] Amir Yazdanbakhsh, Michael Brzozowski, Behnam Khaleghi, Soroush Ghodrati, Kambiz Samadi, Nam Sung Kim, and Hadi Esmaeilzadeh. Flexigan: An end-to-end solution for fpga acceleration of generative adversarial networks. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM18)*, 2018.

[97] Amir Yazdanbakhsh, Kambiz Samadi, Nam Sung Kim, and Hadi Esmaeilzadeh. Ganax: A unified mimd-simd acceleration for generative adversarial networks. In *Proceedings of the 45th Annual International Symposium on Computer Architecture*, pages 650–661. IEEE Press, 2018.

[98] Jiecao Yu, Andrew Lukefahr, David Palframan, Ganesh Dasika, Reetuparna Das, and Scott Mahlke. Scalpel: Customizing DNN pruning to the underlying hardware parallelism. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA)*, pages 548–560, 2017.

[99] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. Optimizing FPGA-based accelerator design for deep convolutional neural networks. In *2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, pages 161–170, 2015.

[100] Chen Zhang, Guangyu Sun, Zhenman Fang, Peipei Zhou, Peichen Pan, and Jason Cong. Caffeine: Towards uniformed representation and acceleration for deep convolutional neural networks. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2018.

[101] Chi Zhang and Viktor Prasanna. Frequency domain acceleration of convolutional neural networks on CPU-FPGA shared memory system. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '17, pages 35–44, New York, NY, USA, 2017. ACM.

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

[103] Yiwei Zhang, Chao Wang, Lei Gong, Yuntao Lu, Fan Sun, Chongchong Xu, Xi Li, and Xuehai Zhou. A power-efficient accelerator based on fpgas for lstm network. In *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 629–630. IEEE, 2017.

[104] Ritchie Zhao, Weinan Song, Wentao Zhang, Tianwei Xing, Jeng-Hau Lin, Mani Srivastava, Rajesh Gupta, and Zhiru Zhang. Accelerating binarized convolutional neural networks with software-programmable fpgas. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 15–24. ACM, 2017.

[105] Aojun Zhou, Anbang Yao, Yiwen Guo, Lin Xu, and Yurong Chen. Incremental network quantization: Towards lossless cnns with low-precision weights. *arXiv preprint arXiv:1702.03044*, 2017.

[106] Julian G. Zilly, Rupesh Kumar Srivastava, Jan Koutník, and Jürgen Schmidhuber. Recurrent highway networks. *CoRR*, abs/1607.03474, 2016.

[107] Wei Zuo, Yun Liang, Peng Li, Kyle Rupnow, Deming Chen, and Jason Cong. Improving high level synthesis optimization opportunity through polyhedral transformations. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '13, pages 9–18, New York, NY, USA, 2013. ACM.