Architecture Design for Highly Flexible and Energy-Efficient Deep Neural Network Accelerators

by

Yu-Hsin Chen

B.S. in EE, National Taiwan University (2009)
S.M. in EECS, Massachusetts Institute of Technology (2013)

Submitted to the
Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy
at the
MASSACHUSETTS INSTITUTE OF TECHNOLOGY
June 2018

Author . . . . . .
Department of Electrical Engineering and Computer Science
May 23, 2018

Certified by . . . . . .
Vivienne Sze
Associate Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Certified by . . . . . .
Joel Emer
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by . . . . . .
Leslie A. Kolodziejski
Professor of Electrical Engineering and Computer Science
Chairman, Department Committee on Graduate Theses

Architecture Design for Highly Flexible and Energy-Efficient Deep Neural Network Accelerators

by

Yu-Hsin Chen

Submitted to the Department of Electrical Engineering and Computer Science
on May 23, 2018, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

Deep neural networks (DNNs) are the backbone of modern artificial intelligence (AI). However, due to their high computational complexity and diverse shapes and sizes, dedicated accelerators that can achieve high performance and energy efficiency across a wide range of DNNs are critical for enabling AI in real-world applications. To address this, we present Eyeriss, a co-design of software and hardware architecture for DNN processing that is optimized for performance, energy efficiency and flexibility. Eyeriss features a novel Row-Stationary (RS) dataflow to minimize data movement when processing a DNN, which is the bottleneck of both performance and energy efficiency. The RS dataflow supports highly-parallel processing while fully exploiting data reuse in a multi-level memory hierarchy to optimize for the overall system energy efficiency given any DNN shape and size. It achieves 1.4× to 2.5× higher energy efficiency than other existing dataflows.

To support the RS dataflow, we present two versions of the Eyeriss architecture. Eyeriss v1 targets large DNNs that have plenty of data reuse. It features a flexible mapping strategy for high performance and a multicast on-chip network (NoC) for high data reuse, and further exploits data sparsity to reduce processing element (PE) power by 45% and off-chip bandwidth by up to 1.9×. Fabricated in a 65nm CMOS, Eyeriss v1 consumes 278 mW at 34.7 fps for the CONV layers of AlexNet, which is 10× more efficient than a mobile GPU. Eyeriss v2 addresses support for the emerging compact DNNs that introduce higher variation in data reuse. It features a RS+ dataflow that improves PE utilization, and a flexible and scalable NoC that adapts to the bandwidth requirement while also exploiting available data reuse. Together, they provide over 10× higher throughput than Eyeriss v1 at 256 PEs. Eyeriss v2 also exploits sparsity and SIMD for an additional 6× increase in throughput.

Thesis Supervisor: Vivienne Sze
Title: Associate Professor of Electrical Engineering and Computer Science

Thesis Supervisor: Joel Emer
Title: Professor of Electrical Engineering and Computer Science

Acknowledgments

First of all, I would like to thank Professor Vivienne Sze and Professor Joel Emer for their guidance and support in the past five years. Project Eyeriss was truly an amazing journey, and I am really grateful to be a part of it. But none of this could have happened if it were not for their complete dedication to the project and my success. They gave me the advice and confidence to go out and be proud of my work, and I could not have asked for better advisors. I would also like to thank my thesis committee member, Professor Daniel Sanchez, for serving on my committee and providing invaluable feedback.

Many contributions in this thesis were the results of collaborations. I would like to thank Tushar Krishna for his support on the implementation of Eyeriss v1, which was so critical that the chip would not have existed without his help. Thanks to Tien-Ju Yang for being the neural network guru in the EEMS group that I could always consult with. His work on sparse DNN also had a direct impact on the design of Eyeriss v2. Thanks to Michael Price and Mehul Tikekar for their tremendous help during chip tapeout.

Thanks to Robin Emer for coming up with the great name of Eyeriss!

Thanks to all members of the EEMS group, including Amr, James, Zhengdong, Jane, Gladynel and Nellie, who have accompanied me for the past five years and shared with me their knowledge, support and friendship.

Thanks to Janice Balzer for always being so helpful with logistics stuff.

Thanks to my wife, Ting-Yun Huang, for always being by my side and providing the help I need throughout my time at MIT. I'm still a functional human being despite my crazy working schedule all because of her. I'd also like to thank my friends at MIT for sharing all the joys and simply being very awesome.

Last but not least, I would like to thank my parents and family for their unconditional support. The journey of PhD is only possible because of them.

This work was funded by DARPA, CICS, a gift from Intel and Nvidia.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Deep neural networks (DNNs) are the cornerstone of modern artificial intelligence (AI) [39]. Their rapid advancement in the past decade is one of the greatest technological breakthroughs in the 21st century, and has enabled machines to deal with many challenging tasks at an unprecedented accuracy, such as speech recognition [16], image recognition [28, 36], and even playing very complex games [59]. While DNNs were proposed back in the 1960s, it was not until the 2010s, when high-performance hardware and a large amount of training data became available, that the superior accuracy and wide applicability of DNNs became broadly received. Since then, it has sparked an AI revolution that impacts numerous research fields and industry sectors with a rapidly growing number of potential use cases.

The next frontier in this AI revolution is to deploy DNNs into real-world applications. However, this also brings new challenges to the existing hardware systems and infrastructure. Conventional general-purpose processors no longer provide satisfying performance[1] (i.e., processing throughput) and energy efficiency for many emerging applications, such as autonomous vehicles, smart assistants, robotics, etc. For example, in 2016, Jeff Dean said "If everyone in the future speaks to their Android phone for three minutes a day...they (Google) would need to double or triple their global computational footprint" [40]. This clearly points out the growing demands on the hardware for AI applications, specifically the

[1] In this thesis, we use the terms *throughput* and *performance* interchangeably, which we define as the number of operations per second or the number of inferences per second, depending on the context. Performance is a commonly used term in the computer architecture community while throughput is a commonly used term in the circuit design community.

fact that dedicated hardware is crucial for meeting the computational needs and lowering the operational cost. The high demands have since spurred the development of dedicated DNN accelerators [45].

However, unlike many standardized technologies, such as video coding or wireless communication protocols, DNNs are still evolving at a very fast pace. Even for the same application there exists a wide range of DNN models that have high variation in their sizes and configurations. Since there is no guarantee on which DNNs are applied at runtime, the hardware should not be designed to target only a limited set of DNNs. Accordingly, in addition to performance and energy efficiency, flexibility is also a very important factor for the design of dedicated DNN accelerators.

In this thesis, we present an accelerator architecture designed for DNN processing, named Eyeriss, that is optimized for performance, energy efficiency and flexibility. Given any DNN model, the hardware has to adapt to its specific configurations and optimize accordingly for performance and energy efficiency. This is achieved through the co-design of the DNN processing dataflow and the hardware architecture. The rest of this chapter will provide the background required to understand the details of this work. Specifically, Section 1.1 provides an overview of DNNs and describes the challenges in DNN processing. Section 1.2 introduces the spatial architecture, which is a commonly used compute paradigm for DNN acceleration. Section 1.3 then discusses prior work in related fields. Finally, Section 1.4 summarizes the contributions of this thesis.

1.1 Overview of Deep Neural Networks

1.1.1 The Basics

DNNs are the realization of the concept of deep learning, which is a part of the broader field of AI as shown in Fig. 1-1. They are inspired by how biological nervous systems communicate and process information. In a DNN, the raw sensory input data is hierarchically transformed, either in time or space, into high-level abstract representations in order to extract useful information. This transformation, called inference, involves multiple stages

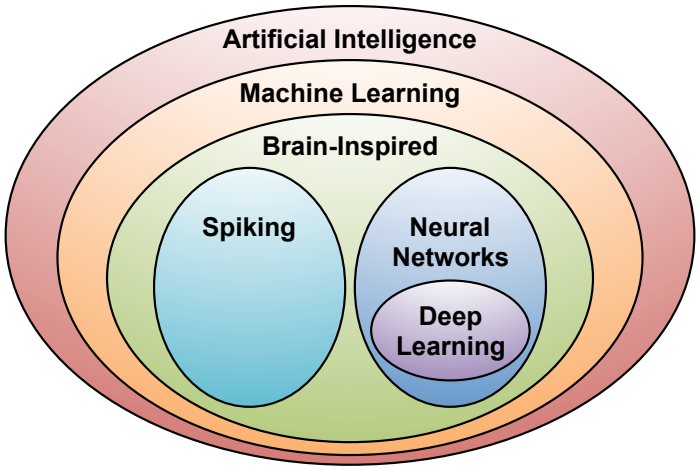
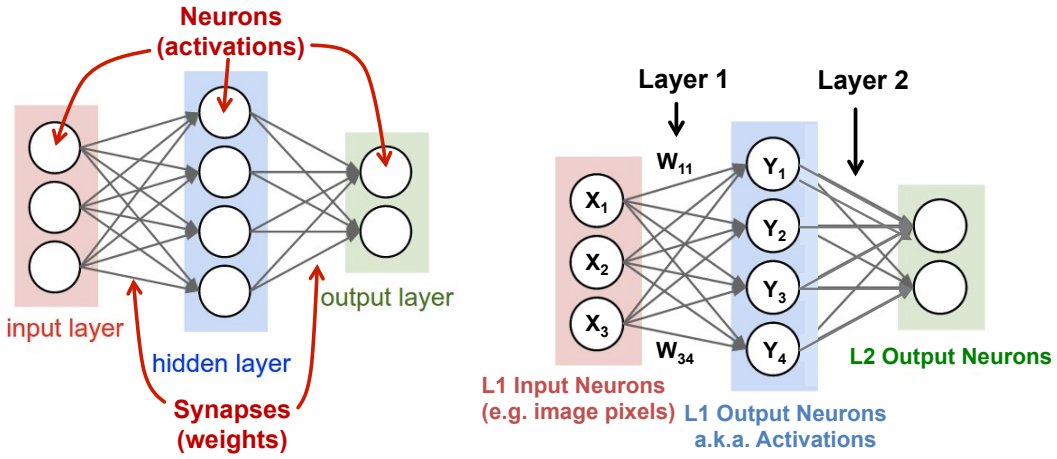


Figure 1-1: The various branches in the field of artificial intelligence.



(a) Neurons and synapses

(b) Compute weighted sums for each layer

Figure 1-2: A simple neural network example and terminology (Figures adopted from [41]).

of non-linear processing, each of which is often referred to as a layer. Fig. 1-2 shows an example of a simple neural network with 2 layers, and a DNN often has from dozens to even thousands of layers. Each DNN layer performs a weighted sum of the input values, called input activations, followed by an non-linear function (e.g., sigmoid, hyperbolic tangent, or the rectified linear unit (ReLU) [48]) as shown in Fig. 1-3. The weights of each layer are determined through a process called training. In this thesis, we will focus on the use of the trained model, which is called inference.

DNNs come in a wide variety of configurations in both the types of layer and their *shapes and sizes*. They are also evolving rapidly with improved accuracy and hardware performance.

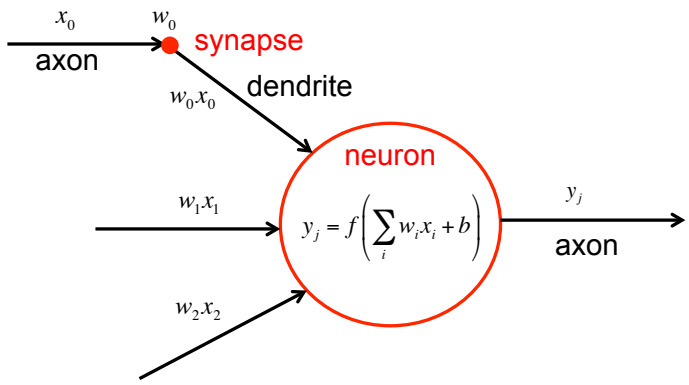


Figure 1-3: The weighted sum in a neuron. x_i, w_i, $f(\cdot)$, and b are the activations, weights, non-linear function and bias, respectively (Figures adopted from [41]).

Despite the many options, there are two primary types of layers that are indispensible in most DNNs: the convolutional (CONV) layer and fully-connected (FC) layer. DNNs that are composed solely of FC layers are also referred to as multi-layer perceptrons (MLP), while DNNs that contain CONV layers are called convolutional neural networks (CNNs).

The processing of both the CONV and FC layers can be described as performing high-dimensional convolutions as shown in Fig. 1-4. In this computation, the input activations of a layer are structured as a set of 2-D *input feature maps* (fmaps), each of which is called a *channel*. Each channel is convolved with a distinct 2-D filter from the stack of filters, one for each channel; this stack of 2-D filters is often referred to as a single 3-D filter. The intermediate values generated by the many 2-D convolutions for each output activation, called partial sums (psums), are then summed across all of the channels. In addition, a 1-D bias can be added to the filtering results, but some recent networks [28] remove its usage from parts of the layers. The results of this computation are the output activations that comprise one channel of *output feature map*. Additional 3-D filters can be used on the same input fmaps to create additional output channels. Finally, multiple input fmaps may be processed together as a *batch* to potentially improve reuse of the filter weights.

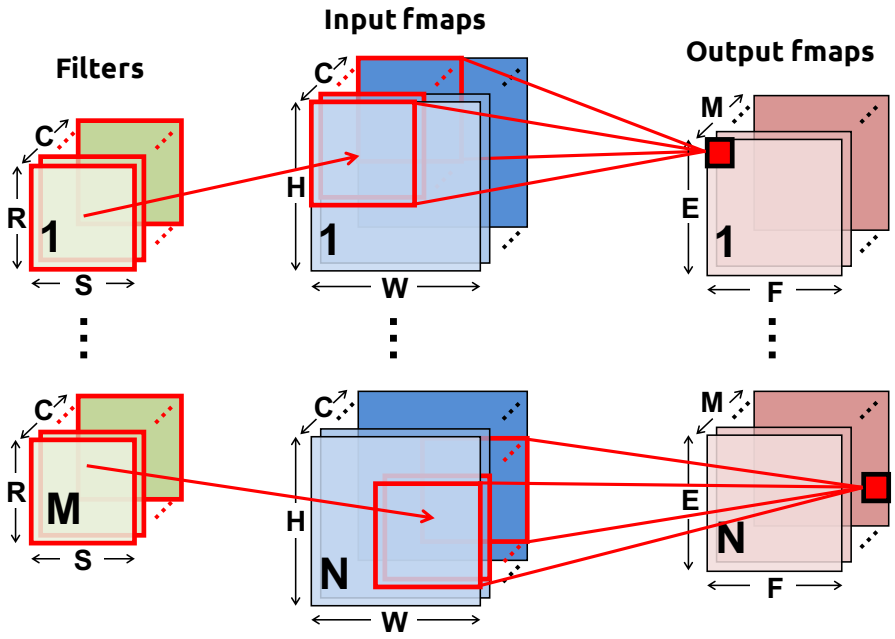


Figure 1-4: The high-dimensional convolution performed in a CONV or FC layer of a DNN.

Given the shape parameters in Table 1.1, the computation of a CONV layer is defined as

$$\mathbf{O}[g][n][m][y][x] = \mathbf{B}[g][m] + \sum_{c=0}^{C-1} \sum_{i=0}^{R-1} \sum_{j=0}^{S-1} \mathbf{I}[g][n][c][Uy+i][Ux+j] \times \mathbf{W}[g][m][c][i][j],$$

$$0 \leq g < G, 0 \leq n < N, 0 \leq m < M, 0 \leq x < F, 0 \leq y < E,$$

$$E = (H - R + U)/U, F = (W - S + U)/U. \tag{1.1}$$

O, **I**, **W** and **B** are the matrices of the output fmaps, input fmaps, filters and biases, respectively. U is a given stride size. For FC layers, Eq. (1.1) still holds with a few additional constraints on the shape parameters: $H = R$, $F = S$, $E = F = 1$, and $U = 1$. Fig. 1-4 shows a visualization of this computation (ignoring biases) assuming $G = 1$, and Fig. 1-5 shows an naive loop nest implementation of this computation.

Shape Parameter	Description
G	number of convolution groups
N	batch size of 3-D fmaps
M	# of 3-D filters / # of output channels
C	# of input channels
H/W	input fmap plane height/width
R/S	filter plane height/width (= H or W in FC)
E/F	output fmap plane height/width (= 1 in FC)

Table 1.1: Shape parameters of a CONV/FC layer.

```
Input Fmaps:    I[G][N][C][H][W]
Filter Weights: W[G][M][C][R][S]
Biases:         B[G][M]
Output Fmaps:   O[G][N][M][E][F]

for (g=0; g<G; g++) {
    for (n=0; n<N; n++) {
        for (m=0; m<M; m++) {
            for (x=0; x<F; x++) {
                for (y=0; y<E; y++) {
                    O[g][n][m][y][x] = B[g][m];
                    for (j=0; j<S; j++) {
                        for (i=0; i<R; i++) {
                            for (k=0; k<C; k++) {
                                O[g][n][m][y][x] +=
                                   I[g][n][k][Uy+i][Ux+j] × W[g][m][k][i][j];
}}}}}}}}
```

Figure 1-5: A naive loop nest implementation of the high-dimensional convolution in Eq. (1.1).

1.1.2 Challenges in DNN processing

In most of the widely used DNNs, the multiply-and-accumulate (MAC) operations in the CONV and FC layers account for over 99% of the overall operations. Not only is the amount of operations high, which can go up to several hundred millions of MACs per layer, they also generate a large amount of data movement. Therefore, they have a significant impact on the processing throughput and energy efficiency of DNNs. Specifically, they pose two challenges: *data handling* and *adaptive processing*. The details of each are described below.

Data Handling: Although the MAC operations in Eq. (1.1) can run at high parallelism, which greatly benefits throughput, it also creates two issues. First, naïvely reading inputs for all MACs directly from DRAM requires high bandwidth and incurs high energy consumption.

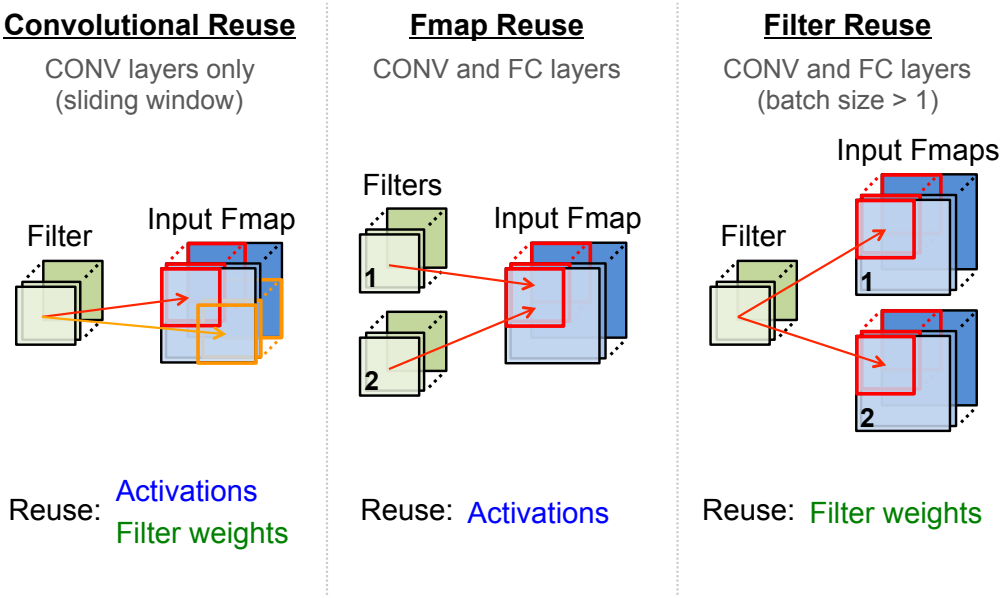


Figure 1-6: Data reuse opportunities in a CONV or FC layers of DNNs [7].

Second, a significant amount of intermediate data, i.e., psums, is generated by the parallel MACs simultaneously, which poses storage pressure and consumes additional memory read and write energy if not processed, i.e., accumulated, immediately.

Fortunately, the first issue can be alleviated by exploiting different types of *input data reuse* as shown in Fig. 1-6:

- **convolutional reuse**: Due to the weight sharing property in CONV layers, a small amount of unique input activations can be shared across many operations. Each filter weight is reused E^2 times in the same input fmap plane, and each input activation is usually reused R^2 times in the same filter plane. FC layers, however, do not have this type of data reuse.

- **filter reuse**: Each filter weight is further reused across the batch of N input fmaps in both CONV and FC layers.

- **fmap reuse**: Each input activation is further reused across M filters (to generate the M output channels) in both CONV and FC layers.

The second issue can be handled by proper *operation scheduling* so that the generated

Layer	H[1]	R/S	E/F	G	C	M	U
CONV1	227	11	55	1	3	96	4
CONV2	31	5	27	2	48	256	1
CONV3	15	3	13	1	256	384	1
CONV4	15	3	13	2	192	384	1
CONV5	15	3	13	2	192	256	1
FC1	6	6	1	1	256	4096	1
FC2	1	1	1	1	4096	4096	1
FC3	1	1	1	1	4096	1000	1

[1] This is the padded size

Table 1.2: CONV/FC layer shape configurations in AlexNet [34].

psums can be reduced as soon as possible to save both the storage space and memory read and write energy. CR^2 psums are reduced into one output activation.

Unfortunately, maximum input data reuse *cannot* be achieved simultaneously with immediate psum reduction, since the psums generated by MACs using the same weight or input activation are not reducible. In order to achieve high throughput and energy efficiency, the underlying DNN *dataflow*, which dictates how the MAC operations are scheduled for processing (Chapter 2 will provide a more formal definition of a dataflow), needs to account for both input data reuse and psum accumulation scheduling at the same time.

Adaptive Processing: The many shape parameters shown in Table 1.1 give rise to many possible CONV and FC layer shapes. Even within the same DNN model, each layer can have distinct shape configurations. Table 1.2 shows the shape configurations of AlexNet [36] as an example. The hardware architecture, therefore, cannot be hardwired to process only certain shapes. Instead, the dataflow must be efficient for different shapes, and the hardware architecture must be programmable to dynamically map to an efficient dataflow.

1.1.3 DNN vs. Conventional Digital Signal Processing

Before DNNs became mainstream, there was already research on high-efficiency and high-performance digital signal processing. For example, convolution processors have a wide applicability in image signal processing (ISP) [55]; linear algebra libraries, such as ATLAS [69], have also been used extensively across many fields. They have proposed

many optimization techniques, including tiling strategies used in multiprocessors and SIMD instructions, to perform convolution or matrix multiplication at high performance on various compute platforms. While many of these techniques can be applied for DNN processing, they do not directly optimize for the best performance and energy efficiency for the following reasons:

- The filter weights in DNNs are obtained through training instead of fixed in the processing system. Therefore, they can consume significant I/O bandwidth and on-chip storage, sometimes comparable to that of the input fmaps.
- Lowering the high-dimensional convolution into matrix multiplication also loses the opportunities to optimize for convolutional data reuse, which can be the key to achieving high energy efficiency.
- The ISP techniques are developed mainly for 2D convolutions. They do not optimize processing resources for data reuse nor do they address the non-trivial psum accumulation in DNN.

1.2 Spatial Architecture

Spatial architectures are a class of accelerators that can exploit high compute parallelism using direct communication between an array of relatively simple processing engines (PEs). They can be designed or programmed to support different algorithms, which are mapped onto the PEs using specialized dataflows. Compared with SIMD/SIMT architectures, spatial architectures are particularly suitable for applications whose dataflow exhibits producer-consumer relationships or can leverage efficient data sharing among a region of PEs.

Spatial architectures come in two flavors: coarse-grained spatial architectures that consist of tiled arrays of ALU-style PEs connected together via on-chip networks [27, 44, 46], and fine-grained spatial architectures that are usually in the form of an FPGA. The expected performance advantage and large design space of coarse-grained spatial architectures has inspired much research on the evaluation of its architectures, control schemes, operation scheduling and dataflow models [2, 22, 49, 51, 58, 63].

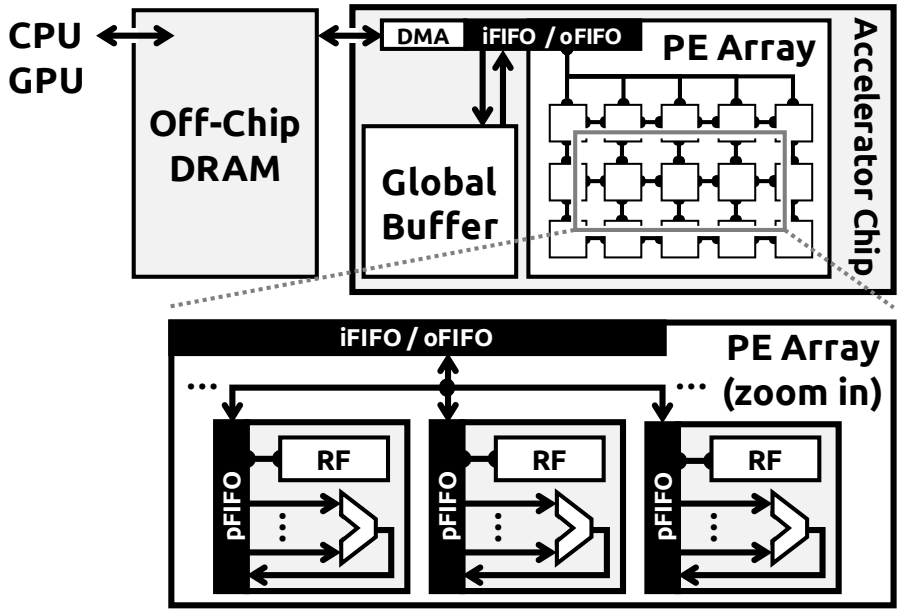


Figure 1-7: Block diagram of a general DNN accelerator system consisting of a spatial architecture accelerator and an off-chip DRAM. The zoom-in shows the high-level structure of a processing element (PE).

Coarse-grained spatial architectures are currently a very popular implementation choice for specialized DNN accelerators for two reasons. First, the operations in a DNN layer are uniform and exhibit high parallelism, which can be computed quite naturally with parallel ALU-style PEs. Second, direct inter-PE communication can be used very effectively for (1) passing partial sums to achieve spatially distributed accumulation, or (2) sharing the same input data for parallel computation without incurring higher energy data transfers. Third, the multi-level storage hierarchy provides many inexpensive ways for data access to exploit data reuse. ASIC implementations usually deploy dozens to hundreds of PEs and specialize the PE datapath only for DNN computation [3, 5, 13, 18, 53]. FPGAs are also used to build DNN accelerators, and these designs usually use integrated DSP slices to construct the PE datapaths [4, 20, 23, 54, 57, 62, 74]. However, the challenge in either type of design lies in the exact mapping of the DNN dataflow to the spatial architecture, since it has a strong impact on the resulting throughput and energy efficiency.

Fig. 1-7 illustrates the high-level block diagram of the accelerator system that is used in this thesis for DNN processing. It consists of a spatial architecture accelerator and off-chip

DRAM. The inputs can be off-loaded from the CPU or GPU to DRAM and processed by the accelerator. The outputs are then written back to DRAM and further interpreted by the main processor.

The spatial architecture accelerator is primarily composed of a global buffer (GLB) and an array of PEs. The DRAM, global buffer and PE array communicate with each other through the input and output FIFOs (iFIFO/oFIFO). The global buffer can be used to exploit input data reuse and hide DRAM access latency, or for the storage of intermediate data. Currently, the typical size of the global buffer used for DNN acceleration is around several hundred kB to a few MB. The PEs in the array are connected via an on-chip network (NoC), and the NoC design depends on the dataflow requirements. The PE includes an ALU datapath, which is capable of doing MAC and addition, a register file (RF) as a local scratchpad (SPad), and a PE FIFO (pFIFO) used to control the traffic going in and out of the ALU. Different dataflows require a wide range of RF sizes, ranging from zero to a few hundred bytes. Typical RF size is below 1kB per PE. Overall, the system provides four levels of storage hierarchy for data accesses, including DRAM, global buffer, NoC (inter-PE communication) and RF. Accessing data from a different level also implies a different energy cost, with the highest cost at DRAM and the lowest cost at RF.

1.3 Related Work

There is currently a large amount of work on the acceleration of DNN processing for various compute platforms, and it is still growing rapidly given the popularity of the research field. Therefore, this section serves to provide a taste of the breadth of research in this field with a representative list of related previous work.

First of all, there is a wide range of proposed architectures for DNN acceleration [3, 4, 5, 13, 18, 20, 23, 35, 47, 50, 53, 54, 57, 60, 62, 73, 74]. While many of them can be described as based on a spatial architecture, it is usually very hard to analyze and compare them due to the following reasons. First, they are often implemented or simulated with different process technologies and available hardware resources. Second, many of them do not report the performance and/or energy efficiency based on publicly available benchmarks. In order

to fairly compare different architectures, we propose (1) a taxonomy of DNN processing dataflows that can capture the essence of how different architectures perform the processing (Chapter 2), and (2) analysis methodologies that can quantify the impact of various dataflows on energy efficiency (Section 3.2). We will then introduce a novel dataflow, called Row Stationary (RS), that can optimize for the overall energy efficiency of the system for vairous DNN shapes and sizes in Chapter 3.

In addition to the analysis on energy efficiency, performance modeling is also a critical part in the design of DNN accelerators. Chen et al. [74] explore various optimization techniques, such as loop tiling and transformation, to map a DNN workload onto an FPGA, and then uses a roofline model [70] of the charateristics of the FPGA to identify the solution with best performance and lowest resource requirements. Jouppi et al. [35] also use a roofline model to illustrate the impact of limited bandwidth on performance. In this work, we propose a method called Eyexam to *tighten the bounds of the roofline model* based on the various architectural design choices and their interaction with the given DNN workload. We also adapt the roofline model for DNN processing by accounting for the fact that different data types (i.e., input activations, weight and psums) will have different bandwidth requirements and thus require three separate roofline models. These techniques will also be discussed in Section 5.2.

As the development of DNNs evolve, many DNN accelerators also start to take advantage of certain properties of the DNN [12]. For example, the activations exhibit a certain degree of sparsity thanks to the ReLU function; also, weight pruning has been shown as an effective method to further reduce the size and computation of a DNN [25, 26]. As a result, architectures such as EIE [24], CNVLUTIN [1] and SCNN [52] have been proposed to take advantage of data sparsity to improve processing throughput and energy efficiency. In Chapter 4 and Chapter 6, we will also discuss how the Eyeriss architecture can exploit data sparsity.

Finally, as we will describe in Chapter 5, the requirement for flexibility has been increasing due to the more diverse set of DNNs proposed in recent years. Previous work that explored flexible hardware for DNNs include FlexFlow [42], DNA [68] and Maeri [38], which propose methods to support multiple dataflows within the same NoC. Rather than

supporting multiple dataflows, Eyeriss proposes using a single but highly flexible dataflow. Along with a highly flexible NoC that can adapt to a wide range of bandwidth requirements while still being able to exploit available data reuse, they can efficiently support a wide range of layer shapes to maximize the number of active PEs and keep the PEs busy for high performance.

1.4 Thesis Contributions

In this thesis, we demonstrate how to optimize for performance and energy efficiency in the processing of a wide range of DNNs. While the processing can benefit from highly-parallel architectures to achieve high throughput, the computation requires a large amount of data, which involves significant data movement that has become the bottleneck of both performance and energy efficiency [14, 29]. To address this issue, the key is to find a *dataflow* that can fully exploit data reuse in the local memory hierarchy and parallelism to minimize accesses to the high-cost memory levels, such as DRAM. The dataflow has to perform this optimization for a wide range of DNN shapes and sizes. In addition, it also has to fully utilize the parallelism to achieve high performance.

1.4.1 Dataflow Taxonomy

Numerous previous efforts have proposed solutions for DNN acceleration. These designs reflect a variety of trade-offs between performance, energy efficiency and implementation complexity. Though with their differences in low-level implementation details, we find that many of them can be described as embodying a set of dataflows. As a result, we are able to classify them into a taxonomy of four dataflow categories. Comparing different implementations in terms of dataflow provides a more objective view of the architecture, and makes it easier to isolate specific contributions in individual designs. This work is discussed in Chapter 2 and appears in [7, 8].

1.4.2 Energy and Performance Analysis Methodologies

We developed methodologies to systematically analyze the energy efficiency and performance of any architectures for DNN processing. They provide a fast way to assess an architecture by examining how the underlying dataflow exploits data reuse in a memory hierarchy and parallelism and how the hardware supports the dataflow. Through these analyses, we have identified certain properties that are critical to the design of DNN accelerators. For example, optimizing for the data reuse of a certain data type does not necessarily translate to the best overall system energy efficiency. Also, to support a diverse set of DNNs, the hardware has to be able to adapt to a wide range of data reuse; otherwise, it can result in low utilization of the parallelism or insufficient data bandwidth to support the processing. We will address these issues in our proposed designs. The energy efficiency analysis framework is discussed in Section 3.2 and appears in [7]. The performance analysis framework is discussed in Section 5.2 in [9].

1.4.3 Energy-efficient Dataflow: Row-Stationary

While the existing dataflows in the taxonomy are popular among many implementations, they are designed to optimize the energy efficiency of accessing a certain data type or memory level in the hierarchy. We propose a new dataflow, called Row-Stationary (RS), that optimizes for the overall system energy efficiency directly by fully exploiting data reuse in a multi-level memory hierarchy for all data types while supporting highly-parallel computation for any given DNN shape and size. The RS dataflow poses the operation mapping process as an optimization problem instead of hand-crafting which data type or memory level should receive the most reuse within the limited hardware resources. It has shown up to 2.5$\times$ higher energy efficiency than other dataflows. This work is discussed in Chapter 3 and appears in [7].

1.4.4 Eyeriss v1 Architecture

We designed an architecture, named Eyeriss v1, to support the RS dataflow and further optimize the hardware for higher performance and energy efficiency. Eyeriss v1 targets large

DNN models, which have plenty of data reuse opportunities for optimization. In addition to the efficiency brought by the RS dataflow, Eyeriss v1 has the following features.

- It uses a flexible mapping strategy to turn the logical mapping in the RS dataflow, which is done regardless of the actual size of the processing element (PE) array in the hardware, into a mapping that fits in the physical dimensions of the PE array. This ensures as many PEs can be utilized as possible for higher performance.

- It uses a multicast on-chip network (NoC) to fully exploit data reuse when delivering data from the global buffer to the PEs. At the same time, the implementation of the multicast NoC shuts down unused data buses to reduce data traffic, which provides an over 80% energy savings over a broadcast NoC design.

- It further exploits the sparsity (zero data) in the feature maps to achieve higher energy efficiency. In particular, it utilizes a zero-skipping logic in the PE to reduce the switching activity of the circuits and the accesses to the register file when data is zero, and saves 45% of PE power. It also compresses the feature map data with a simple run-length coding, which reduces off-chip data bandwidth by 1.2× to 1.9×.

Eyeriss v1 was fabricated in a 65nm CMOS process, and can process the CONV layers of AlexNet at 34.7 fps while consuming 278 mW. The overall energy efficiency was 10× higher than a mobile GPU. We have further integrated the chip with the Caffe deep learning framework and demonstrated an image classification system to showcase the flexibility of the chip for supporting real-world applications. This work is discussed in Chapter 4 and appears in [10, 11].

1.4.5 Highly-Flexible Dataflow and On-Chip Network

The recent trend of DNN development has an increasing focus on reducing the size and computation complexity of DNNs. However, this also results in a higher variation in the amount of data reuse. Many of the existing DNN accelerators were designed for DNNs with plenty of data reuse. As a result, they cannot adapt well to emerging DNNs and therefore lose performance. To solve this problem, we propose two architectural improvements:

- a flexible dataflow, named RS Plus (RS+), that inherits the capability of the RS dataflow to optimize for overall energy efficiency and further improves the utilization of PEs when data reuse is low by being able to parallelize the computation in any dimensions of the data.

- a flexible and scalable hierarchical mesh NoC that can provide high bandwidth when data reuse is low while still being able to exploit high data reuse when available.

Together, they increase the utilization of the PEs in terms of both the number of active PEs and the percentage of active cycles for each PE. Overall, they provide a throughput speedup of over 10× than Eyeriss v1 at 256 PEs, and the performance advantage goes higher when the architecture scales (i.e., increase number of PEs). This work is discussed in Chapter 5 and appears in [12].

1.4.6 Eyeriss v2 Architecture

Eyeriss v2 is designed to support the RS+ dataflow and the hierarchical mesh NoC. In addition, it has the following features:

- it exploits data sparsity to not only improve energy efficiency, but also the processing throughput. This is done by processing data directly in a compressed format for both the feature maps and weights. When data sparsity is low, however, it can still adapt back to process data in the raw format so it does not introduce overhead in the data movement due to compression.

- it introduces SIMD processing in each PE by having two MAC datapaths instead of one. This improves not only throughput but also energy efficiency due to the amortized cost of the memory control logic.

These additional features can bring an additional 6× speedup in throughput. Overall, Eyeriss v2 achieves a speedup of 40× and 10× with 11.3× and 1.9× higher energy efficiency on AlexNet and MobileNet, respectively, over Eyeriss v1 even at the batch size of 1. This work is discussed in Chapter 6 and appears in [12].

Chapter 2

DNN Processing Dataflows

2.1 Definition

The high-dimensional convolutions in DNNs involve a significant amount of MAC operations, which also generate a significant amount of data movement. As described in Section 1.1.2, the challenges in processing are threefold. First of all, the accelerator has to parallelize the MAC operations so that they efficiently utilize the available compute resources in the hardware for high performance. Second, the operations have to be scheduled in a way that can exploit data reuse in a multi-level storage hierarchy, such as the ones introduced in the spatial architecture, in order to minimize data movement for high energy efficiency. This is done by maximizing the reuse of data in the lower-energy-cost storage levels, e.g., local RF, thus minimizing data accesses to the higher-cost levels, e.g., DRAM. Finally, the hardware has to adapt to a wide range of DNN configurations.

These challenges can be framed as an optimization process that finds the optimal MAC operation *mapping* on the hardware architecture for processing. For each MAC operation in a DNN, which is uniquely defined by its associated input activation, weight and psum, the mapping determines its temporal scheduling (in which cycle it is executed) and spatial scheduling (in which PE it is executed) on a highly-parallel architecture. In a mapping optimized for energy efficiency, data in the lower-cost storage levels can be reused by as many MACs as possible before replacement. However, due to the limited amount of local storage, input data reuse (for activations and weights) and local psum accumulation cannot

be fully exploited simultaneously. Therefore, the system energy efficiency is maximized only when the mapping balances all types of data reuse in a multi-level storage hierarchy. In a mapping optimized for throughput, as many PEs will be active for processing for as many cycles as possible, thus minimizing idle compute resources. These two objectives can be further combined to find the mapping that meets a balance of both constraints.

This optimization has to take into account the following two factors: (1) the shape and size of the DNN layer, e.g., number of filters, number of channels, size of filters, etc, which determine the data reuse opportunities, and (2) the available processing parallelism, the storage capacity and the associated energy cost of data access at each level of the memory hierarchy, which are a function of the specific accelerator implementation. Therefore, the optimal mapping changes across different DNN layers as well as hardware configurations.

Due to implementation trade-offs, a specific DNN accelerator design can only find the optimal mapping from the subset of supported mappings instead of the entire mapping space. The subset of supported mappings is usually determined by a set of mapping rules, which also characterizes the hardware implementation. For example, in order to simplify the data delivery from the global buffer to the PE array, the mapping may only allow all parallel MACs to execute on the same weight in order to take advantage of a broadcast NoC design, which sacrifices flexibility for efficiency.

We define the set of mapping rules as a *dataflow* that can be described by a loop nest with pre-defined loop orders and variable loop limits. For simplicity, Fig. 2-1 shows an example dataflow for a 1D convolution between an 1D input fmap of size H and 1D filter of size R, which generates an 1D output fmap of size E. The ordering and parallelization of the for-loops determine the rules for the temporal and spatial scheduling of the MAC operations, respectively. The limit of each loop, e.g., $E2$, $E1$, $E0$, $R2$, $R1$, $R0$, where $E = E2 \times E1 \times E0$, $R = R2 \times R1 \times R0$, however, does *not* have to be determined by the dataflow. This provides flexibility in the dataflow. For example, when $R0 = R$ (and $R1 = 1$ and $R2 = 1$), the processing goes through all weights in the inner-most loop, while when $R2 = R$ (and $R0 = 1$ and $R1 = 1$), the weight stays the same in the inner-most loop. This flexibility creates the supported mapping space for the optimization process to find the optimal mapping for a given objective.

```
Input Fmaps:    I[H]
Filter Weights: W[R]
Output Fmaps:   O[E]

for (e2=0; e2<E2; e2++) {
    for (r2=0; r2<R2; r2++) {
        parallel-for (e1=0; e1<E1; e1++) {
            parallel-for (r1=0; r1<R1; r1++) {
                for (e0=0; e0<E0; e0++) {
                    for (r0=0; r0<R0; r0++) {
                        O[e2*E1*E0+e1*E0+e0] +=
                            I[e2*E1*E0+e1*E0+e0 + r2*R1*R0+r1*R0+r0] ×
                            W[r2*R1*R0+r1*R0+r0];
}}}}}}
```

Figure 2-1: An example dataflow for a 1D convolution.

A mapping is generated by determining the loop limits in a given dataflow, which is a process called *tiling* that is a part of the optimization. Tiling has to take into account not only the shape and size of the data, i.e., H, R and E in the example of Fig. 2-1, but also hardware resources such as the number of PEs, the number of levels in the memory hierarchy and the storage size at each memory level. For example, when only 8 PEs are available, it imposes the following constraint: $E1 \times R1 \leq 8$.

The design of a DNN accelerator architecture starts with the design of the dataflow, and an architecture can be designed to support multiple dataflows at once, which brings the potential advantage of a larger mapping space for optimization. However, there is usually a trade-off between the flexibility and efficiency of the architecture. Since there exists a large number of potential dataflows, it creates an enormous design space for exploration.

2.2 An Analogy to General-Purpose Processors

The operation of DNN accelerators is analogous to that of general-purpose processors as illustrated in Fig. 2-2. In conventional computer systems, the compiler translates the program into machine-readable binary codes for execution; in the processing of DNNs, the mapper translates the DNN shape and size into a hardware-compatible mapping for execution.

Dataflow is a key attribute that is analogous to the architecture of a general-purpose

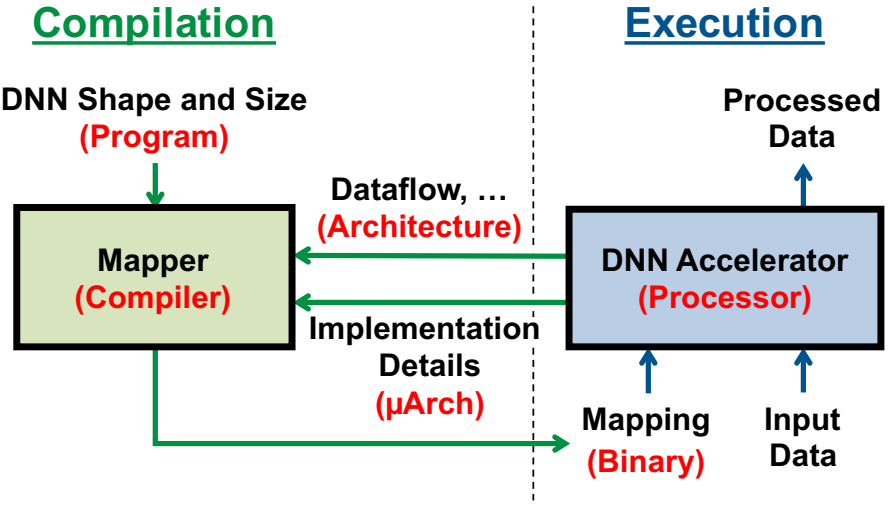


Figure 2-2: An analogy between the operation of DNN accelerators (texts in black) and that of general-purpose processors (texts in red).

processor. Similar to the role of an ISA or memory consistency model, dataflow characterizes the hardware implementation and defines the mapping rules that the mapper has to follow in order to generate hardware-compatible mappings. We consider the dataflow as a part of the architecture, instead of microarchitecture, since we believe it is going to largely remain invariant across implementations. Although, similar to GPUs, the distinction between architecture and microarchitecture is likely to blur for DNN accelerators due to its rapid development. In Section 2.3, we will introduce several existing dataflows that are widely used in implementations.

The hardware implementation details, such as the degree of pipelining in the PE or the bandwidth of NoC for data delivery, are analogous to the microarchitecture of processors for the following reasons: (1) they can vary a lot across implementations, and (2) although they can play a vital part in the optimization, they are not essential since the mapper can always generate sub-optimal mappings.

The goal of the mapper is to search in the mapping space for the best one that optimizes data movement and/or PE utilization. The size of the entire mapping space is determined by the total number of MACs, which can be calculated from the DNN shape and size. However, only a subset of the space is valid given the mapping rules defined by a dataflow. It is the

mapper's job to find out the exact ordering of these MACs on each PE by evaluating and comparing different valid ordering options based on the optimization objective.

As in conventional compilers, performing evaluation is an integral part of the mapper. The evaluation process takes a certain mapping as input and gives an energy consumption and performance estimation based on the available hardware resources (microarchitecture) and the data size and reuse opportunities extracted from the DNN configuration (program). In Section 3.2 and 5.2, we will introduce frameworks that can perform this evaluation.

2.3 A Taxonomy of Existing DNN Dataflows

For computer architects, trade-offs between performance, energy-efficiency and implementation complexity are always of primary concern in architecture designs. This is the same case for designing DNN dataflows. On the one hand, if the dataflow accommodates a large number of valid operation mappings, the mapper has a better chance to find the one with optimal energy efficiency. On the other hand, however, the complexity and cost of hardware implementations might be too high to support such a dataflow, which makes it of no practical use. Therefore, it is very important to identify dataflows that have a strong root in implementation.

Numerous previous efforts have proposed solutions for DNN acceleration. These designs reflect a variety of trade-offs between performance, energy efficiency and implementation complexity. Though with their differences in low-level implementation details, we find that many of them can be described as embodying a set of rules, i.e., *dataflow*, that define the valid mapping space based on how they handle data. As a result, we are able to classify them into a taxonomy of four dataflow categories. In the rest of this section, we will provide a high-level overview on each of the four dataflows.

- **Weight-Stationary (WS) dataflow**: WS keeps filter weights stationary in the RF of each PE by enforcing the following mapping rule: all MACs that use the same filter weight have to be mapped on the same PE for processing contiguously. This maximizes the convolutional and filter reuse of weights in the RF, thus minimizing the energy consumption of accessing weights (e.g., [3, 4, 20, 35, 53]). Fig. 2-3a shows the

data movement of a common WS dataflow implementation. While each weight stays in the RF of each PE, unique input activations are sent to each PE, and the generated psums are then accumulated spatially across PEs.

- **Output-Stationary (OS) dataflow**: OS keeps psums stationary by accumulating them locally in the RF. The mapping rule is: all MACs that generate psums for the same ofmap pixel have to be mapped on the same PE contiguously. This maximizes psum reuse in the RF, thus minimizing energy consumption of psum movement (e.g., [18, 23, 47, 54, 73]). The data movement of a common OS dataflow implementation is to broadcast filter weights while passing unique input activations to each PE (Fig. 2-3b).

- **Input-Stationary (IS) dataflow**: IS keeps input activations stationary in the RF of each PE by enforcing the following mapping rule: all MACs that use the same input activation have to be mapped on the same PE for processing contiguously. This maximizes the convolutional and fmap reuse of input activations in the RF, thus minimizing the energy consumption of accessing input activations (e.g., [52]). In addition to keeping input activations stationary in the RF, a common IS dataflow implementation is to send unique weights to each PE, and the generated psums are then accumulated spatially across PEs (Fig. 2-3c).

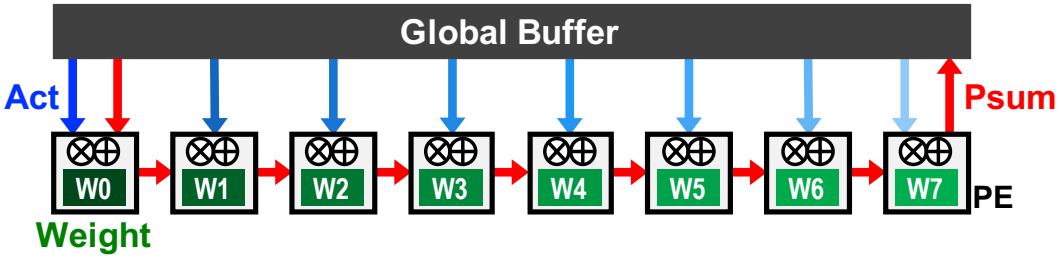
- **No Local Reuse (NLR) dataflow**: Unlike the previous dataflows that keep a certain data type stationary, NLR makes no data stationary locally so it can trade RF off for a large global buffer. This is to minimize DRAM access energy consumption by storing more data on-chip (e.g., [5, 74]). The corresponding mapping rule is: at each processing cycle, all parallel MACs correspond to an unique pair of input channel (for input activation and weight) and output channel (for psum). The data movement of NLR dataflow is to single-cast weights, multi-cast ifmap pixels, and spatially accumulate psums across the PE array (Fig. 2-3d).

The four dataflows show distinct data movement patterns, which imply different trade-offs. First, as is evident in Fig. 2-3a, Fig. 2-3b and Fig. 2-3c, the cost for keeping a specific

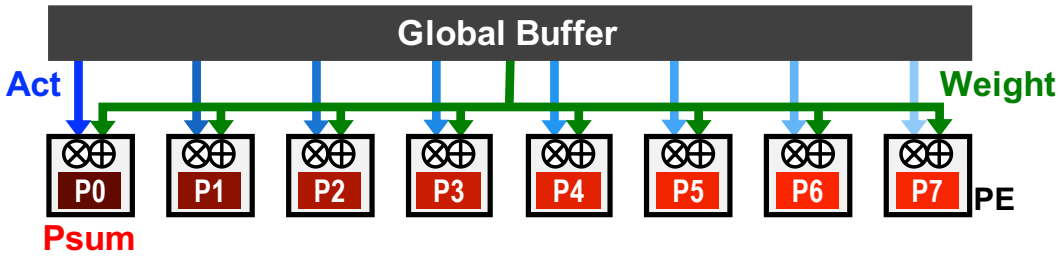
data type stationary is to move the other types of data more. Second, the timing of data accesses also matters. For example, in the OS dataflow, each weight read from the global buffer is broadcast to all PEs with properly mapped MACs on the PE array. This is more efficient than reading the same value multiple times from the global buffer and single-casting it to the PEs, which is the case for filter weights in the NLR dataflow (Fig. 2-3d) due to its mapping restriction. In Chapter 3, we will present a new dataflow that takes these factors into account to optimized for energy efficiency.

2.4 Conclusions

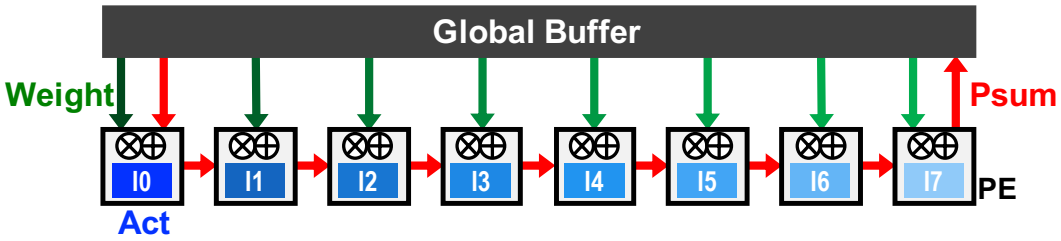
Dataflow is an integral part in the design of DNN accelerators. It dictates the performance and energy efficiency of the hardware, and is a key attribute of the accelerator that is analogous to the architecture of a general-purpose processor. Based on this insight, we find that many of the existing DNN accelerator architectures can be classified into a taxonomy of four dataflows. However, we also notice that these existing dataflows only optimize for the energy efficiency of certain data types or specific levels of memory hierarchy instead of for the overall system energy efficiency. In Chapter 3, we will introduce a new dataflow, called row-stationary, that can achieve this goal.



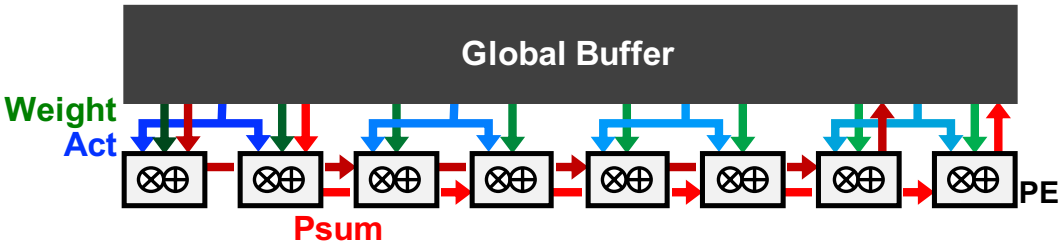
(a) Weight-Stationary (WS) Dataflow



(b) Output-Stationary (OS) Dataflow



(c) Input-Stationary (IS) dataflow



(d) No Local Reuse (NLR) dataflow

Figure 2-3: Taxonomy of existing DNN processing dataflow.

Chapter 3

Energy-Efficient Dataflow: Row-Stationary

3.1 How Row-Stationary Works

While existing dataflows attempt to maximize certain types of input data reuse or minimize the psum accumulation cost, they fail to take all of them into account at once. This results in inefficiency when the layer shape or hardware resources vary. Therefore, it would be desirable if the dataflow could adapt to different conditions and optimize for all types of data movement energy costs. In this chapter, we will introduce a novel dataflow, called *row stationary* (RS) that achieves this goal.

3.1.1 1D Convolution Primitives

The implementation of the RS dataflow in Eyeriss is inspired by the idea of applying a strip mining technique in a spatial architecture [67]. It breaks the high-dimensional convolution down into 1D convolution primitives that can run in parallel; each primitive operates on one row of filter weights and one row of ifmap pixels, and generates one row of psums. Psums from different primitives are further accumulated together to generate the ofmap pixels. The inputs to the 1D convolution come from the storage hierarchy, e.g., the global buffer or DRAM.

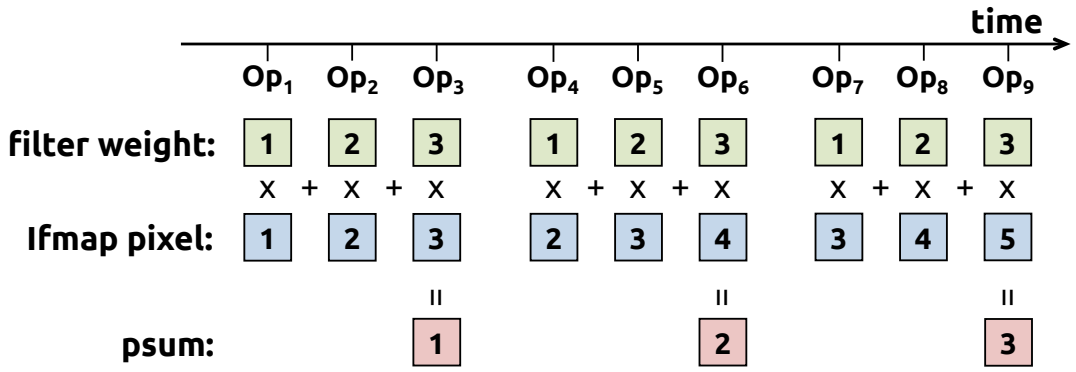


Figure 3-1: Processing of an 1D convolution primitive in the PE. In this example, $R = 3$ and $H = 5$.

Each primitive is mapped to one PE for processing; therefore, the computation of *each row pair stays stationary* in the PE, which creates convolutional reuse of filter weights and ifmap pixels at the RF level. An example of this sliding window processing is shown in Fig. 3-1. However, since the entire convolution usually contains hundreds of thousands of primitives, the exact mapping of all primitives to the PE array is non-trivial, and will greatly affect the energy efficiency.

3.1.2 Two-Step Primitive Mapping

To solve this problem, the primitive mapping is separated into two steps: logical mapping and physical mapping. The logical mapping first deploys the primitives into a logical PE array, which has the same size as the number of 1D convolution primitives and is usually much larger than the physical PE array in hardware. The physical mapping then *folds* the logical PE array so it fits into the physical PE array. Folding implies serializing the computation, and is determined by the amount of on-chip storage, including both the global buffer and local RF. The two mapping steps happen statically prior to runtime, so no on-line computation is required.

Logical Mapping: Each 1D primitive is first mapped to one logical PE in the logical PE array. Since there is considerable spatial locality between the PEs that compute a 2D convolution in the logical PE array, we group them together as a *logical PE set*. Fig. 3-2 shows a logical PE set, where each filter row and ifmap row are horizontally and diagonally

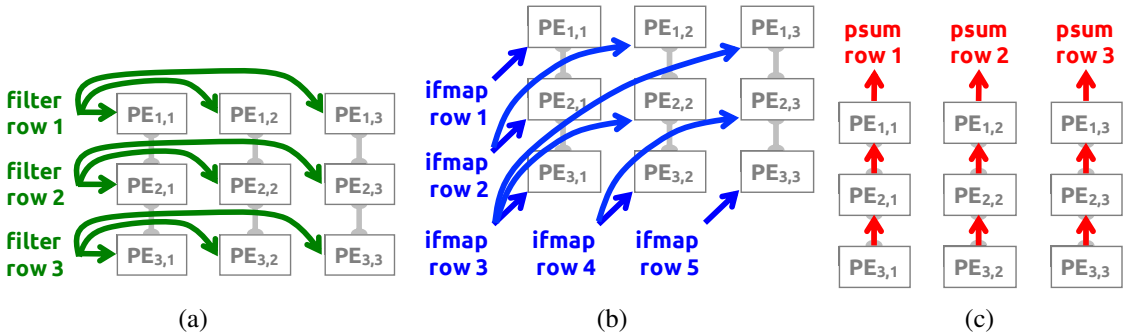


Figure 3-2: The dataflow in a logical PE set to process a 2D convolution. (a) rows of filter weight are reused across PEs horizontally. (b) rows of ifmap pixel are reused across PEs diagonally. (c) rows of psum are accumulated across PEs vertically. In this example, $R = 3$ and $H = 5$.

reused, respectively, and each row of psums is vertically accumulated. The height and width of a logical PE set are determined by the filter height (R) and ofmap height (E), respectively. Since the number of 2D convolutions in a CONV layer is equal to the product of number of ifmap/filter channels (C), number of filters (M) and fmap batch size (N), the logical PE array requires $N \times M \times C$ logical PE sets to complete the processing of an entire CONV layer.

Physical Mapping: Folding means mapping and then running multiple 1D convolution primitives from different logical PEs on the same physical PE. In the RS dataflow, folding is done at the granularity of logical PE sets for two reasons. First, it preserves intra-set convolutional reuse and psum accumulation at the array level (inter-PE communication) as shown in Fig. 3-2. Second, there exists more data reuse and psum accumulation opportunities across the $N \times M \times C$ sets: the same filter weights can be shared across N sets (filter reuse), the same ifmap pixels can be shared across M sets (ifmap reuse), and the psums across each C sets can be accumulated together. Folding multiple logical PEs from the same position of different sets onto a single physical PE exploits input data reuse and psum accumulation at the RF level; the corresponding 1D convolution primitives run on the same physical PE in an interleaved fashion. Mapping multiple sets spatially across the physical PE array also exploits those opportunities at the array level. The exact amount of logical PE sets to fold and to map spatially at each of the three dimensions, i.e., N, M, and C, are determined by the RF size and physical PE array size, respectively. It then becomes an optimization problem to determine the best folding by using the framework in Section 3.2 to evaluate the results.

Section 4.2.4 discusses how this mapping is performed for Eyeriss v1.

After the first phase of folding as discussed above, the physical PE array can process a number of logical PE sets, called a *processing pass*. However, a processing pass still may not complete the processing of all sets in the CONV layer. Therefore, a second phase of folding, which is at the granularity of processing passes, is required. Different processing passes run sequentially on the entire physical PE array. The global buffer is used to further exploit input data reuse and store psums across passes. The optimal amount of second phase folding is determined by the global buffer size, and also requires an optimization using the analysis framework. Section 4.2.5 discusses how the processing passes are scheduled for Eyeriss v1.

3.1.3 Energy-Efficient Data Handling

To maximize energy efficiency, the RS dataflow is built to optimize all types of data movement by maximizing the usage of the storage hierarchy, starting from the low-cost RF to the higher-cost array and global buffer. The way each level handles data is described as follows.

RF: By running multiple 1D convolution primitives in a PE after the first phase folding, the RF is used to exploit all types of data movements. Specifically, there are *convolutional reuse* within the computation of each primitive, *filter reuse* and *ifmap reuse* due to input data sharing between folded primitives, and *psum accumulation* within each primitive and across primitives.

Array (inter-PE communication): *Convolutional reuse* exists within each set and is completely exhausted up to this level. *Filter reuse* and *ifmap reuse* can be achieved by having multiple sets mapped spatially across the physical PE array. *Psum accumulation* is done within each set as well as across sets that are mapped spatially.

Global Buffer: Depending on its size, the global buffer is used to exploit the rest of *filter reuse*, *ifmap reuse* and *psum accumulation* that remain from the RF and array levels after the second phase folding.

3.1.4 Support for Different Layer Types

While the RS dataflow is designed for the processing of high-dimensional convolutions in the CONV layers, it can also naturally support two other layer types:

FC Layer: The computation of FC layers is the same as CONV layers, but without convolutional data reuse. Since the RS dataflow exploits all types of data movement, it can still adapt the hardware resources to cover filter reuse, ifmap reuse and psum accumulation at each level of the storage hierarchy. There is no need to switch between different dataflows as in the case between SOC-MOP and MOC-SOP OS dataflows.

POOL Layer: By swapping the MAC computation with a MAX comparison function in the ALU of each PE, the RS dataflow can also process POOL layers by assuming $N = M = C = 1$ and running each fmap plane separately.

3.1.5 Other Architectural Features

In the Eyeriss architecture, the dataflow in Fig. 3-2 is handled using separate NoCs for the three data types: global multi-cast NoCs for the ifmaps and filters, and a local PE-to-PE NoC for the psums. The architecture can also exploit sparsity by (1) only performing data reads and MACs on non-zero values and (2) compressing the data to reduce data movement. Details on these techniques are described in Section 4.3 and 4.4.3. This brings additional energy savings on top of the efficient dataflow presented in this Chapter.

3.2 Framework for Evaluating Energy Consumption

In order to evaluate the energy efficiency of DNN accelerator architectures implementing various dataflows, in this section, we will introduce a framework for the evaluation of energy consumption of DNN accelerators based on a spatial architecture and its dataflow. The analysis methodology is lightweight yet general, such that is can be applied to the analysis of any DNN accelerator architectures. They are also an indispensable part of the optimization process of finding the optimal mapping for a dataflow.

The way each MAC operation fetches inputs (filter weights and input activations) and accumulates psums introduces different energy costs due to two factors:

- how the dataflow exploits input data reuse and psum accumulation scheduling.
- fetching data from different storage elements in the architecture have different energy costs.

The goal of an energy-efficient dataflow is then to perform most data accesses using the data movement paths with lower energy cost. This is an optimization process that takes all data accesses into account, and will be affected by the layer shape and available hardware resources.

In this section, we will describe a framework that can be used as a tool to optimize the dataflows for spatial architectures in terms of energy efficiency. Specifically, it defines the energy cost for each level of the storage hierarchy in the architecture. Then, it provides a simple methodology to incorporate any given dataflow into an analysis using this hierarchy to quantify the overall data movement energy cost. This allows for a search for the optimal mapping for a dataflow that results in the highest energy efficiency for a given DNN layer shape. It optimizes to maximize reuse of data in the RF, NoC and global buffer.

Data Movement Hierarchy: The spatial architecture provides four levels of storage hierarchy. Sorting their energy cost for data accesses from high to low, it includes DRAM, global buffer, NoC and RF. Fetching data from a higher-cost level to the ALU incurs higher energy consumption. Also, the energy cost of moving data between any of the two levels is dominated by the one with higher cost. Similar to the energy consumption quantification in previous experiments [14, 29, 43], Fig. 3-3 shows the normalized energy consumption of accessing data from each storage level relative to the computation of a MAC at ALU. The numbers are extracted from a commercial 65nm process. The DRAM and global buffer energy costs aggregate the energy of accessing the storage and the iFIFO/oFIFO; the array energy cost includes the energy of accessing the iFIFO/oFIFO/pFIFO on both sides of the path as well as the cost from wiring capacitance.

Analysis Methodology: Given a dataflow, the analysis is formulated in two parts: (1) the input data access energy cost, including filter weights and input activations, and (2) the psum

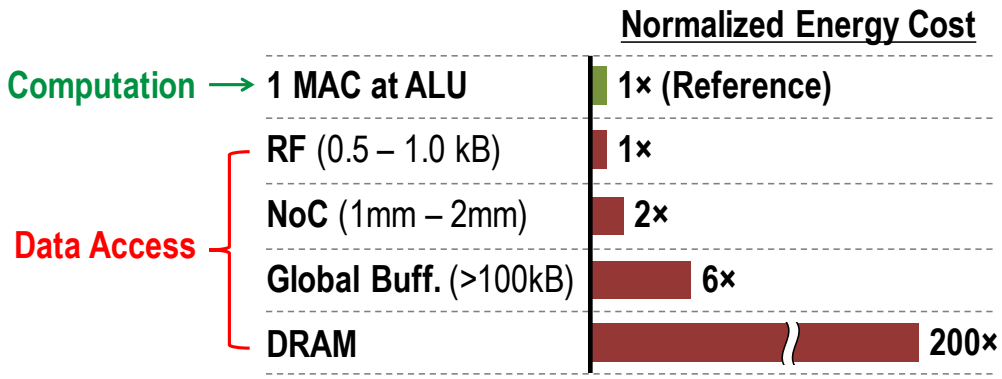


Figure 3-3: Normalized energy cost relative to the computation of one MAC operation at ALU. Numbers are extracted from a commercial 65nm process.

accumulation energy cost. The energy costs are quantified through counting the number of accesses to each level of the previously defined hierarchy, and weighting the accesses at each level with a cost from Fig. 3-3. The overall data movement energy of a dataflow is obtained through combining the results from the two types of input data and the psums.

3.2.1 Input Data Access Energy Cost

If an input data value is reused for many operations, ideally the value is moved from DRAM to RF once, and the ALU reads it from the RF many times. However, due to limited storage and operation scheduling, the data is often kicked out of the RF before exhausting reuse. The ALU then needs to fetch the same data again from a higher-cost level to the RF. Following this pattern, data reuse can be split across the four levels. Reuse at each level is defined as *the number of times each data value is read from this level to its lower-cost levels during its lifetime*. Suppose the total number of reuses for a data value is $a \times b \times c \times d$, it can be split into reuses at DRAM, global buffer, array and RF for a, b, c, and d times, respectively. An example is shown in Fig. 3-4, in which case the total number of reuse, 24, is split into $a = 1$, $b = 2$, $c = 3$ and $d = 4$. The energy cost estimation for this reuse pattern is:

$$\begin{aligned} &a \times EC(\text{DRAM}) + ab \times EC(\text{global buffer}) + \\ &abc \times EC(\text{array}) + abcd \times EC(\text{RF}), \end{aligned} \tag{3.1}$$

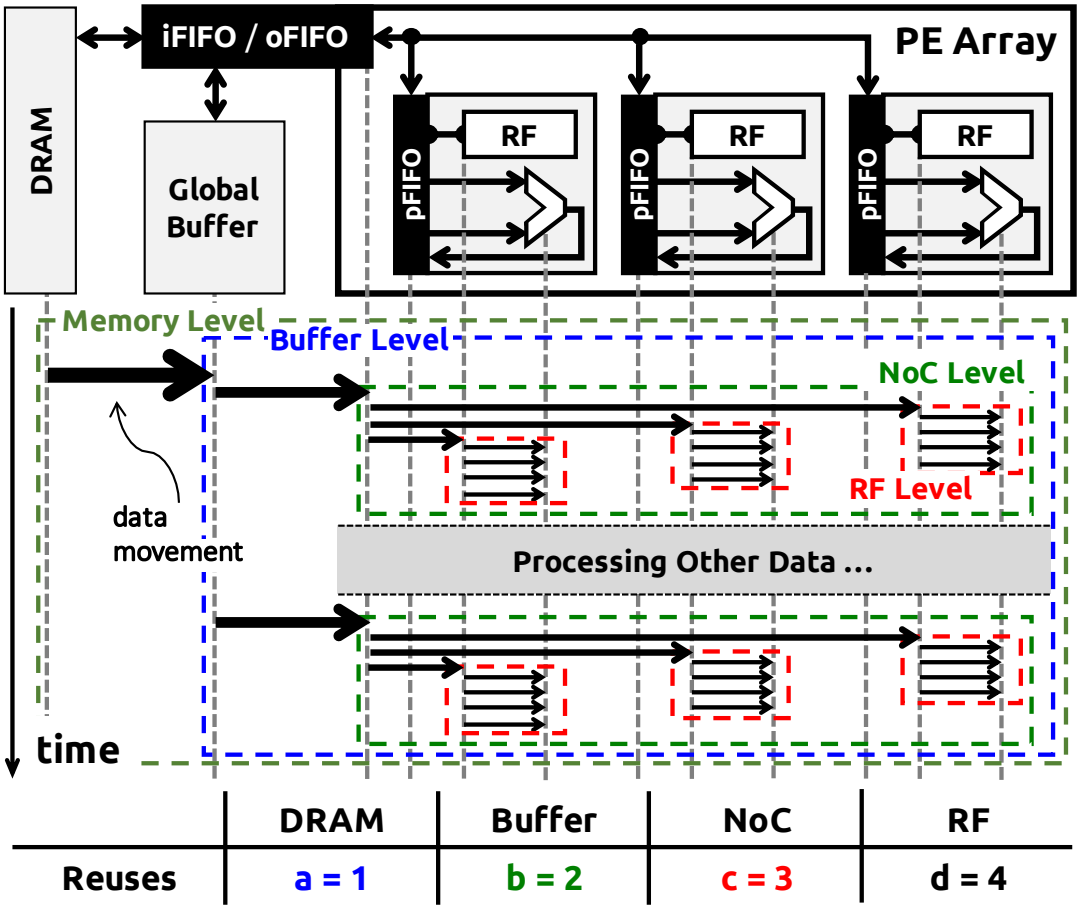


Figure 3-4: An example of the input activation or filter weight being reused across four levels of the memory hierarchy.

where $EC(\cdot)$ is the energy cost from Fig. 3-3. [1]

3.2.2 Psum Accumulation Energy Cost

Psums travel between ALUs for accumulation through the 4-level hierarchy. In the ideal case, each generated psum is stored in a local RF for further accumulation. However, this is often not achievable due to the overall operation scheduling, in which case the psums have to be stored to a higher-cost level and read back again afterwards. Therefore, the total number of accumulations, $a \times b \times c \times d$, can also be split across the four levels. The number

[1] Optimization can be applied to Eq. (3.1) when there is no reuse opportunity. For instance, if $d = 1$, the data is transferred directly from a higher level to the ALU and bypasses the RF, and the last term in Eq. (3.1) can be dropped.

Data Type	Reuse Parameters			
	a	**b**	**c**	**d**
Input Activation	$R/(R0 \times R1 \times R2)$	$R2$	$R1$	$R0$
Weight	$E/(E0 \times E1 \times E2)$	$E2$	$E1$	$E0$
Psums	$R/(R0 \times R1 \times R2)$	$R2$	$R1$	$R0$

Table 3.1: Reuse parameters for the 1D convolution dataflow in Fig. 2-1.

of accumulation at each level is defined as *the number of times each data goes in and out of its lower-cost levels during its lifetime*. An example is shown in Fig. 3-5, in which case the total number of accumulations, 36, is split into $a = 2$, $b = 3$, $c = 3$ and $d = 2$. The energy cost can then be estimated as

$$\begin{aligned}&(2a-1) \times EC(\text{DRAM}) + 2a(b-1) \times EC(\text{global buffer}) + \\ &ab(c-1) \times EC(\text{array}) + 2abc(d-1) \times EC(\text{RF}).\end{aligned} \tag{3.2}$$

The factor of 2 accounts for both reads and writes. Note that in this calculation the accumulation of the bias term is ignored, as it has negligible impact on the overall energy.

3.2.3 Obtaining the Reuse Parameters

For each dataflow, there exists a set of reuse parameters (a, b, c, d) for each of the three data types, i.e., input activations, filter weights and psums, that describes the optimal mapping in terms of energy efficiency under a given DNN layer shape and size. These parameters are a function of the variables in the loop limits of the dataflow, and the optimal values are obtained through an optimization process, currently implemented with the genetic algorithm [21], with objective functions defined in Eq. (3.1) and (3.2). The optimization is constrained by the hardware resources, including the number of PEs and the storage size at each level of the memory hierarchy, and the DNN layer shape and size. For example, the set of reuse parameters for the simple 1D convolution dataflow shown in Fig. 2-1 is a function of $E2$, $E1$, $E0$ and $R2$, $R1$, $R0$, and can be summarized in Table 3.1 (ignoring halos on the edges of the convolution).

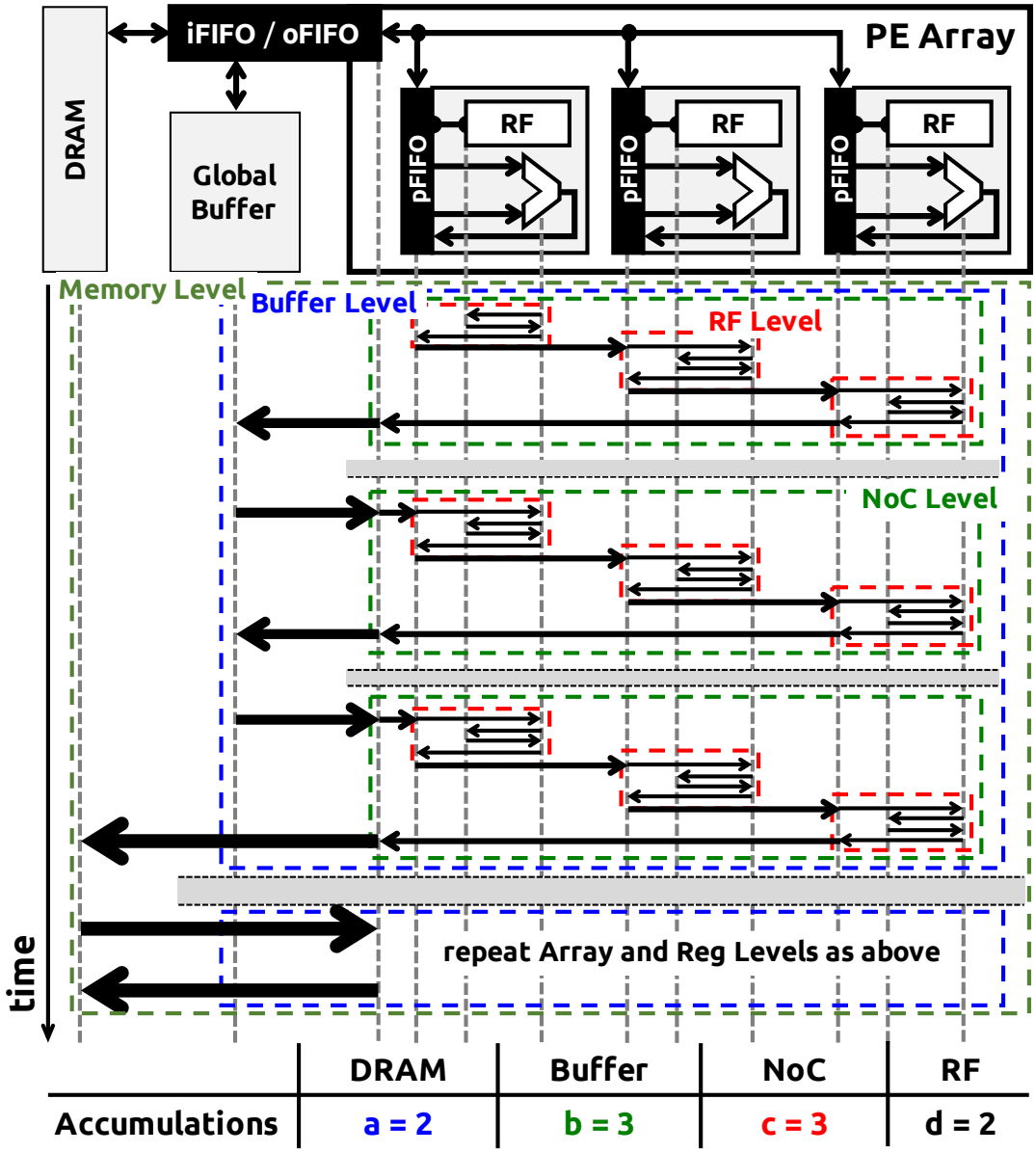


Figure 3-5: An example of the psum accumulation going through four levels of the memory hierarchy.

3.3 Experiment Results

We simulate the RS dataflow and compare its performance with our implementation of existing dataflows (Section 2.3) under the same hardware area and processing parallelism constraints. The mapping for each dataflow is optimized by our framework (Section 3.2) for the highest energy efficiency. AlexNet [36] is used as the CNN model for benchmarking

due to its high popularity. Its 5 CONV and 3 FC layers also provide a wide range of shapes that are suitable for testing the adaptability of different dataflows. The simulations assume 16 bits per word, and the result aggregates data from all CONV or FC layers in AlexNet. To save space, the SOC-MOP, MOC-MOP and MOC-SOP OS dataflows are renamed as OS_A, OS_B and OS_C, respectively.

3.3.1 RS Dataflow Energy Consumption

The RS dataflow is simulated with the following setup: (1) 256 PEs, (2) 512B RF per PE, and (3) 128kB global buffer. Batch size is fixed at 16. Fig. 3-6 shows the energy breakdown across the storage hierarchy in the 5 CONV and 3 FC layers of AlexNet. The energy is normalized to one ALU operation, i.e., a MAC.

The two types of layers show distinct energy distributions. On the one hand, the energy consumption of CONV layers is dominated by RF accesses, which shows that RS fully exploits different types of data movement in the local RF and minimizes accesses to storage levels with higher cost. This distribution is verified by our Eyeriss chip measurement results where the ratio of energy consumed in the RF to the rest (except DRAM) is also roughly 4:1. On the other hand, DRAM accesses dominate the energy consumption of FC layers due to the lack of convolutional data reuse. Overall, however, CONV layers still consume approximately 80% of total energy in AlexNet, and the percentage is expected to go even higher in modern CNNs that have more CONV layers.

3.3.2 Dataflow Comparison in CONV Layers

We compare the RS dataflow with existing dataflows in (1) DRAM accesses, (2) energy consumption and (3) energy-delay product (EDP) using the CONV layers of AlexNet. Different hardware resources (256, 512 and 1024 PEs) and batch sizes (N = 1, 16 and 64) are simulated to further examine the scalability of these dataflows.

DRAM Accesses: DRAM accesses are expected to have a strong impact on the overall energy efficiency since their energy cost is orders of magnitude higher than other on-chip data movements. Fig. 3-7 shows the average DRAM accesses per operation of the 6 dataflows.

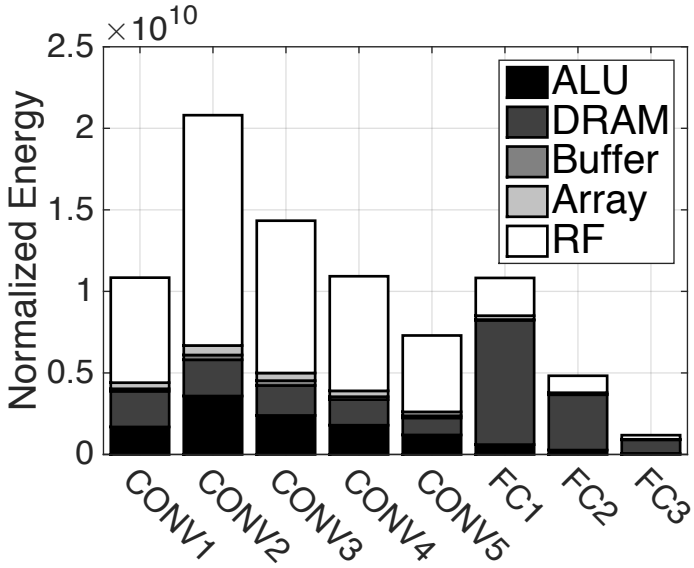


Figure 3-6: Energy consumption breakdown of RS dataflow in AlexNet.

DRAM writes are the same across all dataflows since we assume the accelerator writes only ofmaps but no psums back to DRAM. In this scenario, RS, OS_A, OS_B and NLR have significantly lower DRAM accesses than WS and OS_C, which means the former achieve more data reuse on-chip than the latter. Considering RS has much less on-chip storage compared to others, it shows the importance of co-designing the architecture and dataflow. In fact, RS can achieve the best energy efficiency when taking the entire storage hierarchy into account instead of just DRAM accesses, which will be discussed later in this section.

The WS dataflow is optimized for maximizing weight reuse. However, it sacrifices ifmap reuse due to the limited number of filters that can be loaded on-chip at a time, which leads to high DRAM accesses. The number of filters is limited by (1) insufficient global buffer size for psum storage, and (2) size of PE array. In fact, Fig. 3-7a shows a case where WS cannot even operate due to the global buffer being too small for a batch size of 64. OS_C also has high DRAM accesses since it does not exploit convolutional reuse of ifmaps on-chip.

In terms of architectural scalability, all dataflows can use the larger hardware area and higher parallelism to reduce DRAM accesses. The benefit is most significant on WS and OS_C, which also means that they are more demanding on hardware resources. For batch size scalability, increasing N from 1 to 16 reduces DRAM accesses for all dataflows since it

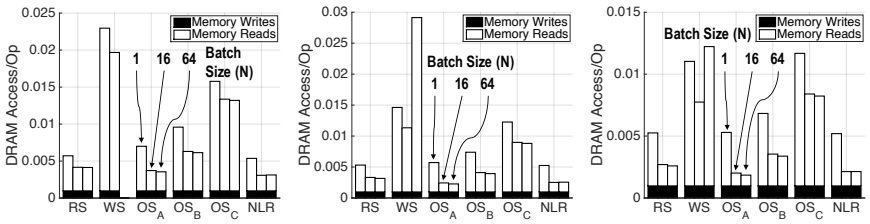


Figure 3-7: Average DRAM accesses per operation of the six dataflows in CONV layers of AlexNet under PE array size of (a) 256, (b) 512 and (c) 1024.

gives more filter reuse, but saturates afterwards. The only exception is WS, which cannot handle large batch sizes well due to the psum storage issue.

Energy Consumption: Fig. 3-8 shows the normalized energy consumption per operation of the 6 dataflows. Overall, RS is 1.4× to 2.5× more energy efficient than other dataflows. Although OS_A, OS_B and NLR have similar or even lower DRAM accesses compared with RS, RS still consumes lower total energy by fully exploiting the lowest-cost data movement at the RF for all data types. The OS and WS dataflows use the RF only for psum accumulation and weight reuse, respectively, and therefore spend a significant amount of energy in the array for other data types. NLR does not use the RF at all. Most of its data accesses come from the global buffer directly, which results in high energy consumption.

Fig. 3-8d shows the same energy result at a PE array size of 1024 but with energy breakdown by data type. The results for other PE array sizes show a similar trend. While the WS and OS dataflows are most energy efficient for weight and psum accesses, respectively, they sacrifice the reuse of other data types: WS is inefficient at ifmap reuse, and the OS dataflows cannot reuse ifmaps and weights as efficiently as RS since they focus on generating psums that are reducible. NLR does not exploit any type of reuse of weights in the PE array, and therefore consumes most of its energy for weight accesses. RS is the only dataflow that optimizes energy for all data types simultaneously.

When scaling up the hardware area and processing parallelism, the energy consumption per operation roughly stays the same across all dataflows except for WS, which sees a decrease in energy due to the larger global buffer size. Increasing batch size helps to reduce energy per operation similar to the trend shown in the case of DRAM accesses. The energy

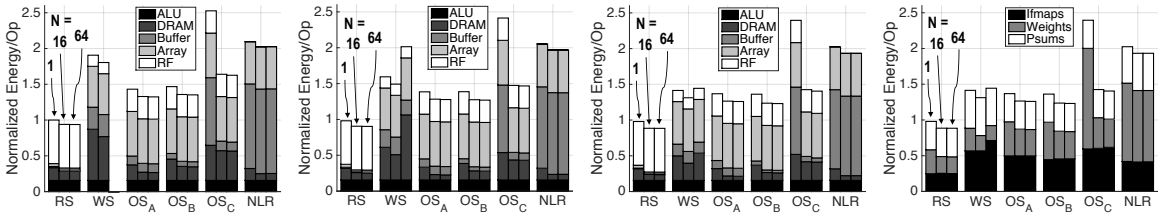


Figure 3-8: Energy consumption of the six dataflows in CONV layers of AlexNet under PE array size of (a) 256, (b) 512 and (c) 1024. (d) is the same as (c) but with energy breakdown in data types. The energy is normalized to that of RS at array size of 256 and batch size of 1. The RS dataflow is 1.4× to 2.5× more energy efficient than other dataflows.

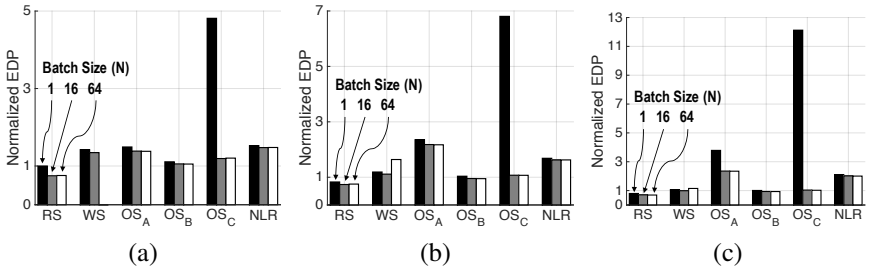


Figure 3-9: Energy-delay product (EDP) of the six dataflows in CONV layers of AlexNet under PE array size of (a) 256, (b) 512 and (c) 1024. It is normalized to the EDP of RS at PE array size of 256 and batch size of 1.

consumption of OS_C, in particular, improves significantly with batch sizes larger than 1, since there is no reuse of weights at RF and array levels when batch size is 1.

Energy-Delay Product: Energy-delay product is used to verify that a dataflow does not achieve high energy efficiency by sacrificing processing parallelism, i.e., throughput. Fig. 3-9 shows the normalized EDP of the 6 dataflows. The delay is calculated as the reciprocal of number of active PEs. A dataflow may not utilize all available PEs due to the shape quantization effects and mapping constraints. For example, when batch size is 1, the maximum number of active PEs in OS_A and OS_C are the size of 2D ofmap plane (E^2) and the number of ofmap channels (M), respectively. Compared with the other dataflows, RS has the lowest EDP since its mapping of 1D convolution primitives efficiently utilizes available PEs. OS_A and OS_C show high EDP at batch size of 1 due to its low PE utilization, especially at larger array sizes.

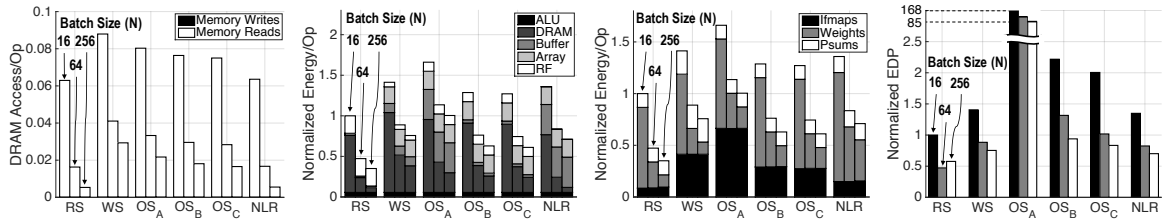


Figure 3-10: (a) average DRAM accesses per operation, energy consumption with breakdown in (b) storage hierarchy and (c) data types, and (d) EDP of the six dataflows in FC layers of AlexNet under PE array size of 1024. The energy consumption and EDP are normalized to that of RS at batch size of 1.

3.3.3 Dataflow Comparison in FC Layers

We run the same experiments as in Section 3.3.2 but with the FC layers of AlexNet. Fig. 3-10 shows the results of 6 dataflows under a PE array size of 1024. The results for other PE array sizes show the same trend. The batch size now starts from 16 since there is little data reuse with a batch size of 1, in which case the energy consumptions of all dataflows are dominated by DRAM accesses for weights and are approximately the same. The DRAM accesses, however, can be reduced by techniques such as pruning and quantization of the values [25].

Compared with existing dataflows, the RS dataflow has the lowest DRAM accesses, energy consumption and EDP in the FC layers. Even though increasing batch size helps to improve energy efficiency of all dataflows due to more filter reuse, the gap between RS and the WS/OS dataflows becomes even larger since the energy of the latter are dominated by ifmap accesses. In fact, OS_A runs FC layers very poorly because its mapping requires ifmap pixels from the same spatial plane, while the spatial size of FC layers is usually very small. Overall, the RS dataflow is at least 1.3× more energy efficient than other dataflows at a batch size of 16, and can be up to 2.8× more energy efficient at a batch size of 256.

3.3.4 Hardware Resource Allocation for RS

For the RS dataflow, we further experiment changing the hardware resource allocation between processing and storage under a fixed area. This is to determine its impact on energy

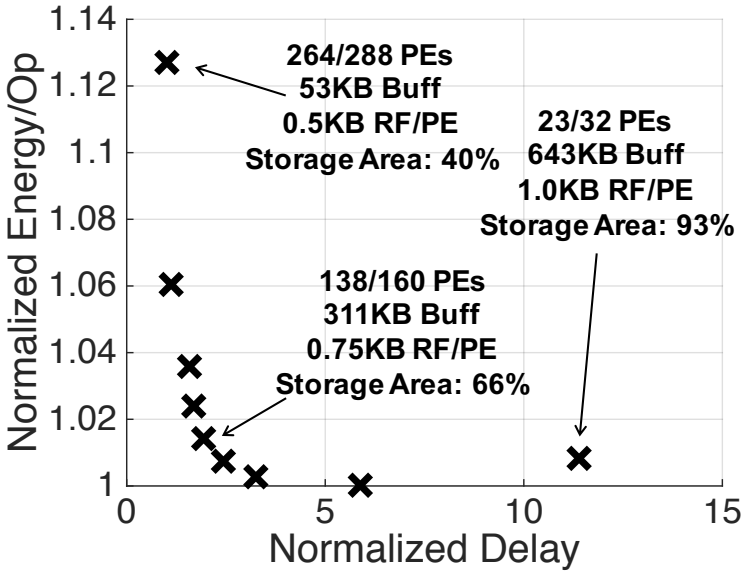


Figure 3-11: Relationship between normalized energy per operation and processing delay under the same area constraint but with different processing area to storage area ratio.

efficiency and throughput. The fixed area is based on the setup using 256 PEs with the baseline storage area. We sweep the number of PEs from 32 to 288 and adjust the size of RF and global buffer to find the lowest energy cost in CONV layers of AlexNet for each setup.

Fig. 3-11 shows the normalized energy and processing delay of different resource allocations. First, although the throughput increases by more than 10× by increasing the number of PEs, the energy cost only increases by 13%. This is because a larger PE array also creates more data reuse opportunities. Second, the trade-off between throughput and energy is not monotonic. The energy cost becomes higher when the PE array size is too small due to (1) there is little data reuse in the PE array, and (2) the global buffer is already large enough that increasing the buffer size does not contribute much to data reuse.

3.4 Conclusions

This chapter presents an analysis framework to evaluate the energy cost of different DNN dataflows on a spatial architecture. It accounts for the energy cost of different levels of the storage hierarchy under fixed area and processing parallelism constraints. It also can be used to search for the most energy-efficient mapping for each dataflow. Under this

framework, a novel dataflow, called row stationary (RS), is presented that minimizes energy consumption by maximizing input data reuse (weights and input activations) and minimizing psum accumulation cost simultaneously, and by accounting for the energy cost of different storage levels. Compared with existing dataflows using AlexNet as a benchmark, the RS dataflow is 1.4× to 2.5× more energy efficient in CONV layers, and at least 1.3× more energy efficient in FC layers for batch sizes of 16 and above.

Chapter 4

Eyeriss v1

In this chapter, we introduce Eyeriss v1, a DNN accelerator architecture built to support the RS dataflow (Chapter 3). We will provide an overview of the architecture in Section 4.1, and then discuss how does the architecture perform the processing with mappings from the RS dataflow in Section 4.2 in a way that maximizes the utilization of the PEs to achieve high performance. Section 4.3 introduces the hardware features that improves energy efficiency by exploiting data statistics. Section 4.4 describes the details in the design of the architectural modules. Finally, we will provide the implementation results in Section 4.5.

4.1 Architecture Overview

Fig. 4-1 shows the top-level architecture and memory hierarchy of the Eyeriss v1 system. It has two clock domains: the core clock domain for processing, and the link clock domain for communication with the off-chip DRAM through a 64-bit bi-directional data bus. The two domains run independently and communicate through an asynchronous FIFO interface. The core clock domain consists of a spatial array of 168 processing elements (PEs) organized as a 12×14 rectangle, an 108KB global buffer (GLB), a run-length compression (RLC) CODEC, and a ReLU module. To transfer data for computation, each PE can either communicate with its neighbor PEs or the GLB through a network-on-chip (NoC), or access a memory space that is local to the PE, called scratchpads (SPads) (Section 4.4.3). Overall, there are four levels of memory hierarchy in the system (in decreasing energy per access): DRAM,

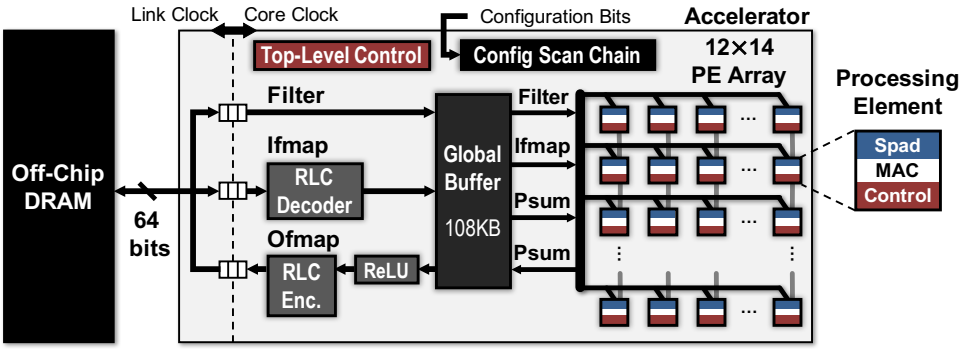


Figure 4-1: Eyeriss v1 top-level architecture.

GLB, NoC, and SPads.

The accelerator has two levels of control hierarchy. The top-level control coordinates (1) traffic between the off-chip DRAM and the GLB through the asynchronous interface, (2) traffic between the GLB and the PE array through the NoC, and (3) operation of the RLC CODEC and ReLU module. The lower-level control consists of control logic in each PE, which runs independently from each other. Therefore, even though the 168 PEs are identical and run under the same core clock, their processing states do not need to proceed in lock steps, i.e., not as a systolic array. Each PE can start its own processing as soon as any data arrives, either activations, weights or psums.

The accelerator runs the processing of a DNN layer-by-layer. For each layer, it first loads the configuration bits into a 1794-bit scan chain serially to reconfigure the entire accelerator, which takes less than 100μs. These bits prepare the accelerator for the processing of a certain DNN layer shape and size, which includes setting up the PE array computation mapping according to the RS dataflow and NoC data delivery patterns (Section 4.4.2). They are generated offline and are statically accessed at runtime. Then, the accelerator loads tiles of the input activations and weights from DRAM for processing, and the computed output activations are written back to DRAM. Batches of input activations for the same layer can be processed sequentially without further reconfigurations of the chip.

4.2 Flexible Mapping Strategy

In this section, we will discuss how does the Eyeriss v1 architecture implement the processing for the mappings of the RS dataflow. We will use AlexNet as an example throughout the discussion.

4.2.1 1D Convolution Primitive in a PE

The RS dataflow first divides the high-dimensional convolutions into 1D convolution primitives, each of which runs on a PE. Due to the sliding window processing of the primitive, each PE can use the local SPads for both convolutional data reuse and psum accumulation. Since only a sliding window of data has to be retained at a time, the required SPad capacity depends only on the filter row size (S) but not the input fmap row size (W), and is equal to: (1) S for a row of filter weights, (2) S for a sliding window of input fmap values, and (3) 1 for the psum accumulation. In AlexNet, for example, possible values for S are 11 (layer CONV1), 5 (layer CONV2) and 3 (layers CONV3–CONV5). Therefore, the minimum SPad capacity required for filter weights, input fmap activations and psum are 11, 11 and 1, respectively, to support all layers.

4.2.2 2D Convolution PE Set

A 2D convolution is composed of many 1D convolution primitives, and its computation also (1) shares the same row of filter or input fmap across primitives, and (2) accumulates the psums from multiple primitives together. Therefore, a *PE Set* is grouped to run a 2D convolution and exploit the inter-primitive convolutional reuse and psum accumulation. In a set, each row of filter is reused horizontally, each row of input fmap is reused diagonally, and rows of psum are accumulated vertically. The dimensions of a PE set are determined by the filter and output fmap size of a given layer. Specifically, the height and width of the PE set are equal to the number of filter rows (R) and ofmap rows (E), respectively. In AlexNet, the PE sets are of size 11×55 (CONV1), 5×27 (CONV2), and 3×13 (CONV3–5).

4.2.3 PE Set Mapping

The dimensions of a PE set are a function of the shape of a layer and are independent of the physical dimensions of the PE array. Therefore, a strategy is required to map these PE sets onto the PE array. This strategy should try to maintain the arrangement of the PEs in the PE set to take advantage of using nearby PEs to share data and accumulate psums in a set. In Eyeriss v1, a PE set can be mapped to any group of PEs in the array that has the same dimensions. However, there are two exceptions:

- *The PE set has more than 168 PEs*: This can be solved by *strip mining* the 2D convolution, i.e., the PE set only processes e rows of ofmap at a time, where $e \leq E$. The dimensions of the strip-mined PE set then becomes $R \times e$ and can fit into the PE array.

- *The PE set has less than 168 PEs, but has width larger than 14 or height larger than 12*: A PE set that is too wide is divided into separated segments that are mapped independently to the array. Eyeriss v1 currently does not support the mapping of a PE set that is taller than the height of the PE array. Therefore, the maximum natively supported filter height is 12.

An example of these two exceptions can be seen from the PE set mapping of layers CONV1–5 in AlexNet onto the 12×14 PE array of Eyeriss v1 as shown in Fig. 4-2. The 11×55 PE set of CONV1 is strip-mined to 11×7. The strip-mined PE set width is determined by a process that optimizes for overall energy efficiency as introduced in [7] and discussed in Section 3.1.2 of this thesis. The 5×27 PE set of CONV2 is divided into two segments with dimensions 5×14 and 5×13, respectively, and each segment is independently mapped onto the PE array. Finally, the 3×13 PE set of CONV3–5 can easily fit into the PE array. Except for CONV2, the PE array can fit multiple PE sets in parallel as shown in Fig. 4-2, and the RS dataflow further defines how to fully utilize hardware resources to minimize data movement in the dimensions beyond 2D. This mapping strategy is realized by a custom designed NoC that is also optimized for energy efficiency (Section 4.4.2).

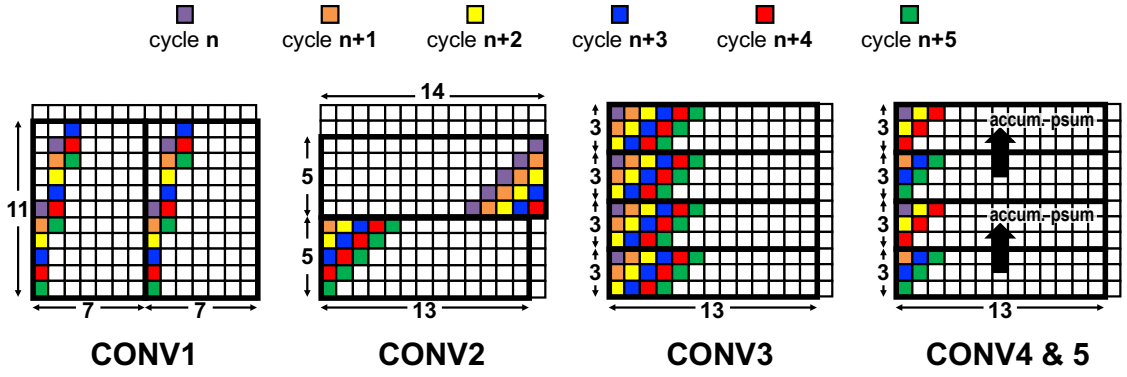


Figure 4-2: Mapping of the PE sets on the spatial array of 168 PEs for the CONV layers in AlexNet. For the colored PEs, the PEs with the same color receive the same input activation in the same cycle. The arrow between two PE sets indicates that their psums can be accumulated together.

4.2.4 Dimensions Beyond 2D in PE Array

Processing of many 2D convolutions are required to complete the high-dimensional convolutions due to the three additional dimensions: batch size (N), number of channels (C), and number of filters (M). Assuming varying only one dimension at a time and fixing rest of the two the same, two 2D convolutions that use (1) different input fmaps reuse the same filter (i.e., filter reuse), (2) different filters reuse the same input fmap (i.e., fmap reuse), (3) filters and input fmaps from different channels can accumulate their psums together (i.e., psum accumulation). The filter reuse can be exploited simply by streaming different input activations through the same PE set sequentially (Fig. 4-3a), since the filter stays constant in the set. The fmap reuse and psum accumulation opportunities can also be exploited by using either the SPads or the spatial parallelism of the PE array, so data accesses to DRAM and the GLB are further reduced.

Multiple 2D Convolutions in a PE Set: If the SPad size is large enough, each PE can run multiple 1D convolution primitives simultaneously by interleaving their computation. Equivalently, this means *each PE set is running multiple 2D convolutions* on different filters and channels. There are two scenarios:

- By interleaving the computation of primitives that run on the same input fmap with different filters, the SPads can buffer the same input activation and reuse it to compute

with a weight from each filter sequentially (Fig. 4-3b). It requires increasing the filter and psum SPad size.

- By interleaving the computation of primitives that run on different channels, the PE can accumulate through all channels sequentially on the same psum (Fig. 4-3c). This requires increasing the input fmap and filter SPad size.

The mapping of multiple primitives in the same PE can be described by parameters p and q. Each PE runs $p \times q$ primitives simultaneously from q different channels of p different filters. The required SPad capacity for each data type is (1) $p \times q \times S$ for the rows of filter weights from q channels of p filters, (2) $q \times S$ for q sliding windows of input activations from q different input channels, and (3) p for the accumulation of psums in p output channels. In Eyeriss v1, where input fmap SPad is 12×16b, filter SPad is 224×16b and psum SPad is 24×16b, p can be up to 24, and q can be up to 4 in AlexNet since the minimum S is 3.

Multiple PE Sets in the PE Array: As shown in Fig. 4-2, the PE array can fit more than one PE set if the set is small enough. Mapping multiple sets has the two extra advantages in addition to the increase on processing throughput: (1) the same input activation is read once from the GLB and reused in multiple sets simultaneously, and (2) the psums from different sets are further accumulated within the PE array directly.

The mapping of multiple sets is described by parameters r and t. The PE array fits $r \times t$ PE sets in parallel that run r different channels of t different filters simultaneously. Every t sets share the same input fmap with t filters, and every r sets that run on r input channels accumulate their psums within the PE array. Fig. 4-2 shows the mapping of multiple sets and the reuse of input activations in Eyeriss v1. Specifically, CONV1 and CONV3 have $t = 2$ and 4, respectively, and the same input activation is sent to all sets. CONV4 and CONV5 have $r = t = 2$. The same input activation is sent to every other set, and the psums from the top and bottom two sets are accumulated together. In each layer, the PEs that are not covered by any sets are clock gated to save energy consumption.

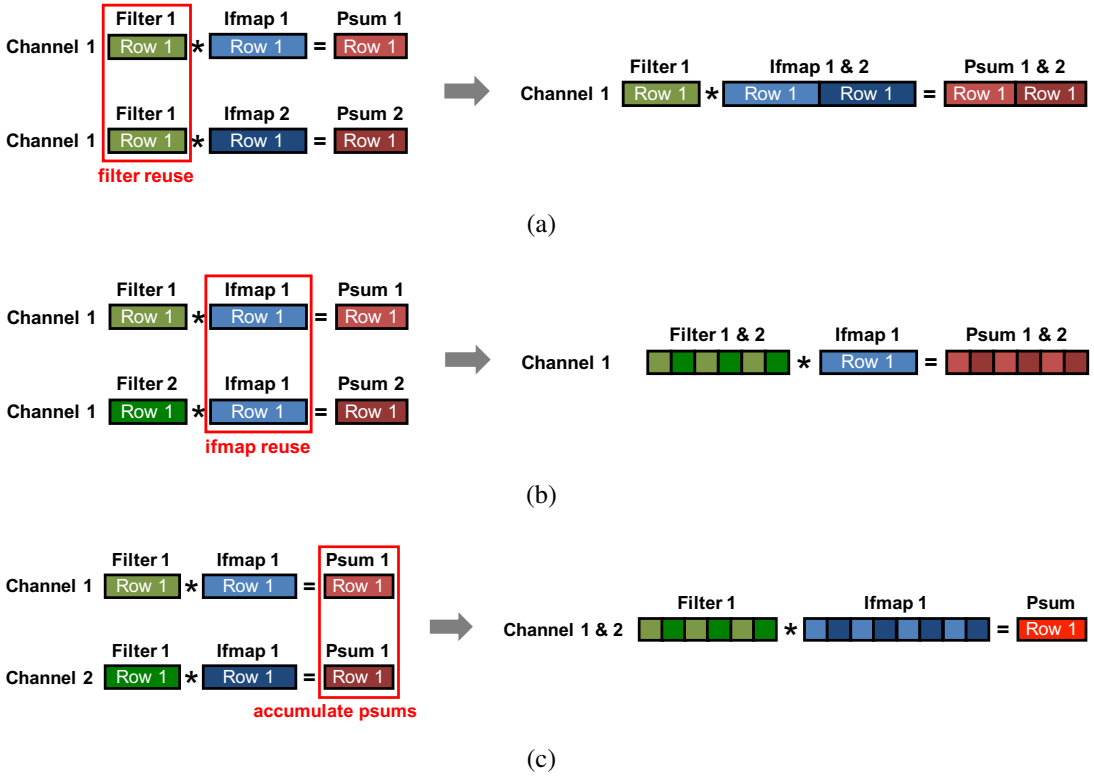


Figure 4-3: Handling the dimensions beyond 2D in each PE: (a) by concatenating the input fmap rows, each PE can process multiple 1D primitives with different input fmaps and reuse the same filter row, (b) by time interleaving the filter rows, each PE can process multiple 1D primitives with different filters and reuse the same input fmap row, (c) by time interleaving the filter and input fmap rows, each PE can process multiple 1D primitives from different channels and accumulate the psums together.

4.2.5 PE Array Processing Passes

So far we have described a way to exploit data reuse by maximally utilizing the storage of SPads and the spatial parallelism of the PE array. The PE array can run multiple 2D convolutions from up to $q \times r$ channels of $p \times t$ filters simultaneously. Multiple input fmaps can also be processed sequentially through the array. The amount of computation done in this fashion is called a *Processing Pass*. In a pass, each input data is read only once from the GLB, and the psums are stored back to the GLB only once when the processing is finished.

A DNN layer usually requires hundreds to thousands of processing passes to complete its processing, and fmap reuse and psum accumulation also exist across these passes. The GLB is used to exploit these opportunities by buffering two types of data: input activations and psums. The input activations stored in the GLB can be reused across multiple processing

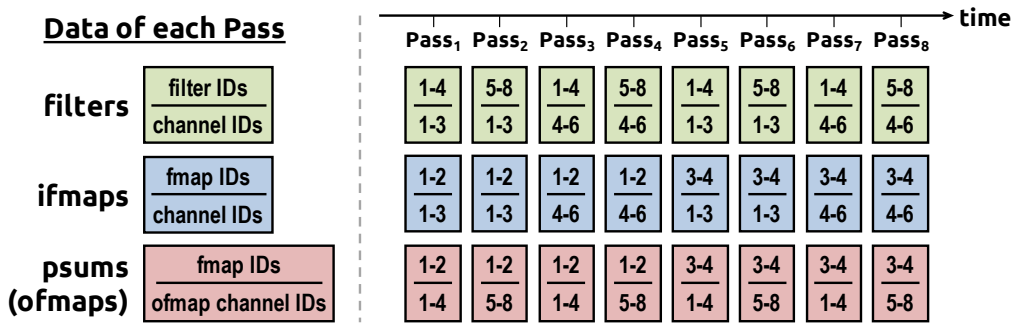


Figure 4-4: Scheduling of processing passes. Each block of filters, input fmaps (ifmaps) or psums is a group of 2D data from the specified dimensions used by a processing pass. The number of channels (C), filters (M) and ifmaps (N) used in this example layer created for demonstration purpose are 6, 8 and 4, respectively, and the RS dataflow uses 8 passes to process the layer.

passes; the psums that are accumulated across passes use the GLB as the intermediate storage, so they do not go off-chip until the final ofmap values are obtained.

Fig. 4-4 shows the scheduling of processing passes. This example layer created only for illustrative purposes has 6 channels (C), 8 filters (M) and 4 input fmaps (N). A pass is assumed to process 3 channels ($q \times r$) and 4 filters ($p \times t$). Also, the batch size that a pass processes, denoted as n, is assumed to be 2. Overall, the computation of this layer uses 8 processing passes. Each group of input fmaps is read from DRAM once, stored in the GLB, and reused in two consecutive passes with total 8 filters to generate 8 ofmap channels. However, this also requires the GLB to store psums from two consecutive passes so they do not go to DRAM. In this case, the GLB needs to store $m = 8$ ofmap channels. Each filter weight is read from DRAM into the PE array once for every 4 passes.

The scheduling of the processing passes determines the storage allocation required for input fmaps and psums in the GLB. Specifically, $n \times q \times r$ 2D input fmaps and $n \times m$ 2D psums have to be stored in the GLB for reuse. Since these parameters change based on the mapping of each layer, the GLB allocation for input fmaps and psums has to be reconfigurable to store them in different proportions.

Summary: Table 4.1 summarizes a list of dataflow mapping parameters that define the mapping of the RS dataflow. For a given DNN shape, these parameters are determined by

Parameter	Description
m	number of output channels stored in the global buffer
n	batch size used in a processing pass
e	width of the PE set (strip-mined if necessary)
p	number of output channels processed by a PE set
q	number of input channels processed by a PE set
r	number of PE sets that process different input channels in the PE array.
t	number of PE sets that process different output channels in the PE array.

Table 4.1: Mapping parameters of the RS dataflow.

Layer	DNN Shape Parameters						RS Mapping Params							GLB Allocation	
	H/W[1]	R/S	E/F	C	M	U	m	n	e	p	q	r	t	ifmap	psum
CONV1	227	11	55	3	96	4	96	1	7	16	1	1	2	15.5KB	72.2KB
CONV2	31	5	27	48	256	1	64	1	27	16	2	1	1	3.8KB	91.1KB
CONV3	15	3	13	256	384	1	64	4	13	16	4	1	4	7.0KB	84.5KB
CONV4	15	3	13	192	384	1	64	4	13	16	3	2	2	10.5KB	84.5KB
CONV5	15	3	13	192	256	1	64	4	13	16	3	2	2	10.5KB	84.5KB

[1] This is the padded size

Table 4.2: The shape parameters of AlexNet and its RS dataflow mapping parameters on Eyeriss v1. This mapping assumes a batch size (N) of 4.

an optimization process that takes (1) the energy cost at each level of the memory hierarchy and (2) the hardware resources, including the GLB size, SPad size and number of PEs, into account [7]. Table 4.2 lists the RS dataflow mapping parameters used for AlexNet in Eyeriss v1. It also shows the storage required in the GLB for both input fmaps and psums.

4.3 Exploit Data Statistics

Even though the RS dataflow optimizes data movement for all data types, the intrinsic amount of data and the corresponding computation are still high. To further improve energy efficiency, data statistics of DNN is explored to (1) compress the DRAM accesses, which is the most energy consuming data movement per access, on top of the optimized dataflow, and (2) skip the unnecessary computation to save processing power (Section 4.4.3).

The ReLU function introduces many zeros in the fmaps by rectifying all negative filtering results to zero. While the number of zeros in the fmaps depends on the input data to the DNN, it tends to increase with deep layers. In AlexNet, almost 40% of ifmap values of CONV2 are zeros on average, and it goes up to around 75% at CONV5. In addition to the

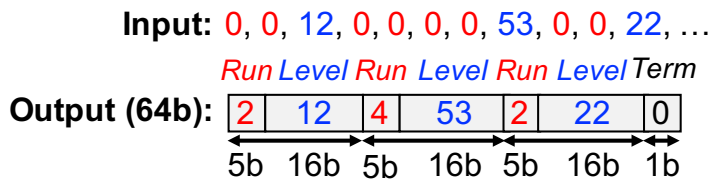


Figure 4-5: Encoding of the Run-length compression (RLC).

fmap, recent studies have also shown that up to 91% of filter weights in a DNN can be pruned to zero [26, 71]; Chapter 6 will discuss how weight sparsity is exploited in Eyeriss v2.

Run-length compression (RLC) is used in Eyeriss v1 to exploit the zeros in fmaps and save DRAM bandwidth. Fig. 4-5 shows an example of RLC encoding. Consecutive zeros with a maximum run length of 31 is represented using a 5-bit number as the *Run*. The next value is inserted directly as a 16-bit *Level*, and the count for run starts again. Every 3 pairs of run and level are packed into a 64-bit word, with the last bit indicating if the word is the last one in the code. Based on our experiments using AlexNet with the ImageNet dataset, The compression rate of RLC only adds 5% to 10% overhead to the theoretical entropy limit.

Except for the input data to the first layer of a DNN, all of the fmaps are stored in RLC compressed form in the DRAM. The accelerator reads the encoded ifmaps from DRAM, decompresses it with the RLC decoder and writes it into the GLB. The computed ofmaps are read from the GLB, processed by the ReLU module optionally, compressed by the RLC encoder and transmitted to the DRAM. This saves both space and R/W bandwidth of the DRAM. From our experiments using AlexNet, the DRAM accesses for fmaps alone, including both ifmaps and ofmaps, can be saved by nearly 30% in CONV1, and nearly 75% in CONV5. Fig. 4-6 shows the overall DRAM accesses in AlexNet before and after RLC. The traffic includes filters, ifmaps and ofmaps. The DRAM access could be further reduced if compression were applied to filter weights.

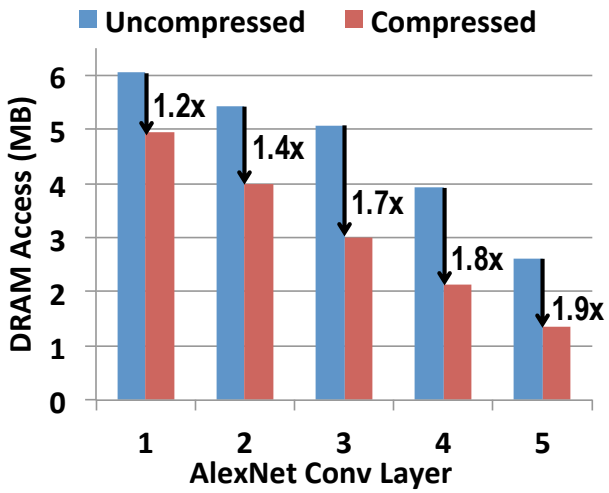


Figure 4-6: Comparison of DRAM accesses, including filters, ifmaps and ofmaps, before and after using RLC in the 5 CONV layers of AlexNet.

4.4 System Modules

4.4.1 Global Buffer

The Eyeriss v1 accelerator has a global buffer (GLB) of 108KB that can communicate with DRAM through the asynchronous interface and with the PE array through the NoC. The GLB stores all three types of data: input activations, weights, and psums/output activations. 100KB of the GLB is allocated for input activations and psums as required by the RS dataflow for reuse. Even though it is not required by the dataflow, the remaining 8KB (two banks of 512x64b SRAMs) of the GLB is allocated for filter weights to compensate for the insufficient off-chip traffic bandwidth of the test setup. While the PE array is working on a processing pass, the GLB preloads the weights used by the next processing pass.

The 100KB storage space for input activations and psums has to be reconfigurable to fit the two data types in different proportions for supporting different shapes (Table 4.2). It also has to provide enough bandwidth for accesses from the PE array. To meet the two demands, the space is divided into 25 banks, each of which is a 512x64b (4KB) SRAM. Each bank is assigned entirely for input activations or psums, and the assignment is reconfigurable based on the scan chain bits. Therefore, the PE array can access both input activations and psums simultaneously, each from one of the 25 banks.

4.4.2 Network-on-Chip

The NoC manages data delivery between the GLB and the PE array as well as between different PEs. The NoC architecture needs to meet the following goals. First, the NoC has to support the data delivery patterns used in the RS dataflow. While the data movement within a PE set is uniform, there are three scenarios in the mapping of real DNNs that can break the uniformity and should be taken care of: (1) different convolution strides (U) result in the input activation delivery skipping certain rows in the array (AlexNet CONV1 in Fig. 4-2), (2) a set is divided into segments that are mapped onto different parts of the PE array (AlexNet CONV2 in Fig. 4-2), and (3) multiple sets are mapped onto the array simultaneously and different data is required for each set (AlexNet CONV4–5 in Fig. 4-2). Second, the NoC should leverage the data reuse achieved by the RS dataflow to further improve energy efficiency. Third, it has to provide enough bandwidth for data delivery in order to support the highly-parallel processing in the PE array.

Conventional approaches usually use hop-by-hop mesh NoC at the cost of increased ramp-up time and router overhead [15, 31]. Therefore, we chose to implement a custom NoC for the required data delivery patterns that is optimized for latency, bandwidth, energy and area. The custom NoC comprises three different types of networks as described below.

Global Input Network: The GIN is optimized for a single-cycle multicast from the GLB to a group of PEs that receive the same filter weight, input activation or psum. Fig. 4-2 shows an example of input activation delivery in AlexNet. The challenge is that the group of destination PEs varies across layers due to the differences in data type, convolution stride, and mapping. Broadcasting each data with a bit-vector tag of the same size of the PE array (i.e., 168 bits), which indicates the IDs of destination PEs, can support any arbitrary mapping. However, doing so is also very costly in terms of both area and energy consumption due to the increased GIN bus width. Instead, we implemented the GIN, as shown in Fig. 4-7, with two levels of hierarchy: Y-bus and X-bus. A vertical Y-bus consists of 12 horizontal X-buses, one at each row of the PE array, and each X-bus connects to 14 PEs in the row. Each X-bus has a *row* ID, and each PE has a *col* ID. These IDs are all *reconfigurable*, and an unique ID is given to each group of X-buses or PEs that receives the same data in a

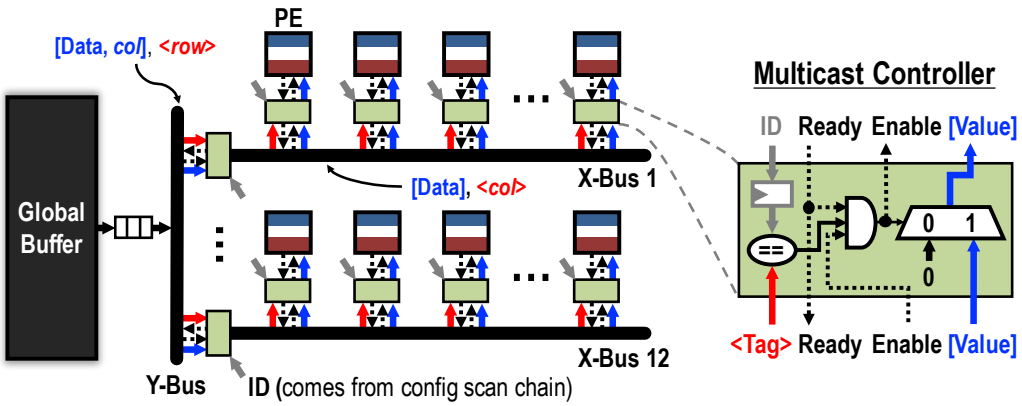


Figure 4-7: Architecture of the global input network (GIN).

given DNN layer. Each data read from the GLB is augmented with a (row, col) tag by the top-level controller, and the GIN guarantees that the data is delivered to *all and only* the X-buses and then PEs with the ID that matches the tag within a single cycle. The tag-ID matching is done using the *Multicast Controller* (MC). There are 12 MCs on the Y-bus, to compare the *row* tag with the *row* ID of each X-bus, and 14 MCs on each of the X-buses to compare the *col* tag with the *col* ID of each PE. The unmatched X-buses and PEs are gated to save energy. For flow control, the data is passed from the GLB down to the GIN only when all destination PEs have issued a ready signal. An example of the *row* and *col* ID setup for input activation delivery using GIN in AlexNet is shown in Fig. 4-8.

Eyeriss v1 has separate GINs for each of the three data types (weights, input activations and psums) to provide sufficient bandwidth from the GLB to the PE array. All GINs have 4-bit *row* IDs to address the 12 rows. The weight and psum GINs use 4-bit *col* IDs to address the 14 columns, while input activation GIN uses 5-bit to support maximum 32 input fmap rows passing in diagonal. The weight and psum GINs have data bus width of 64b (4×16b), while the input activation GIN has the data bus width of 16b.

Global Output Network: The GON is used to read the psums generated by a processing pass from the PE array back to the GLB. The GON has the same architecture as the GIN; only the direction of data transfer is reversed. The data bus width is also 64b as the psum GIN.

Local Network: Between every pair of PEs that are on two consecutive rows of the same

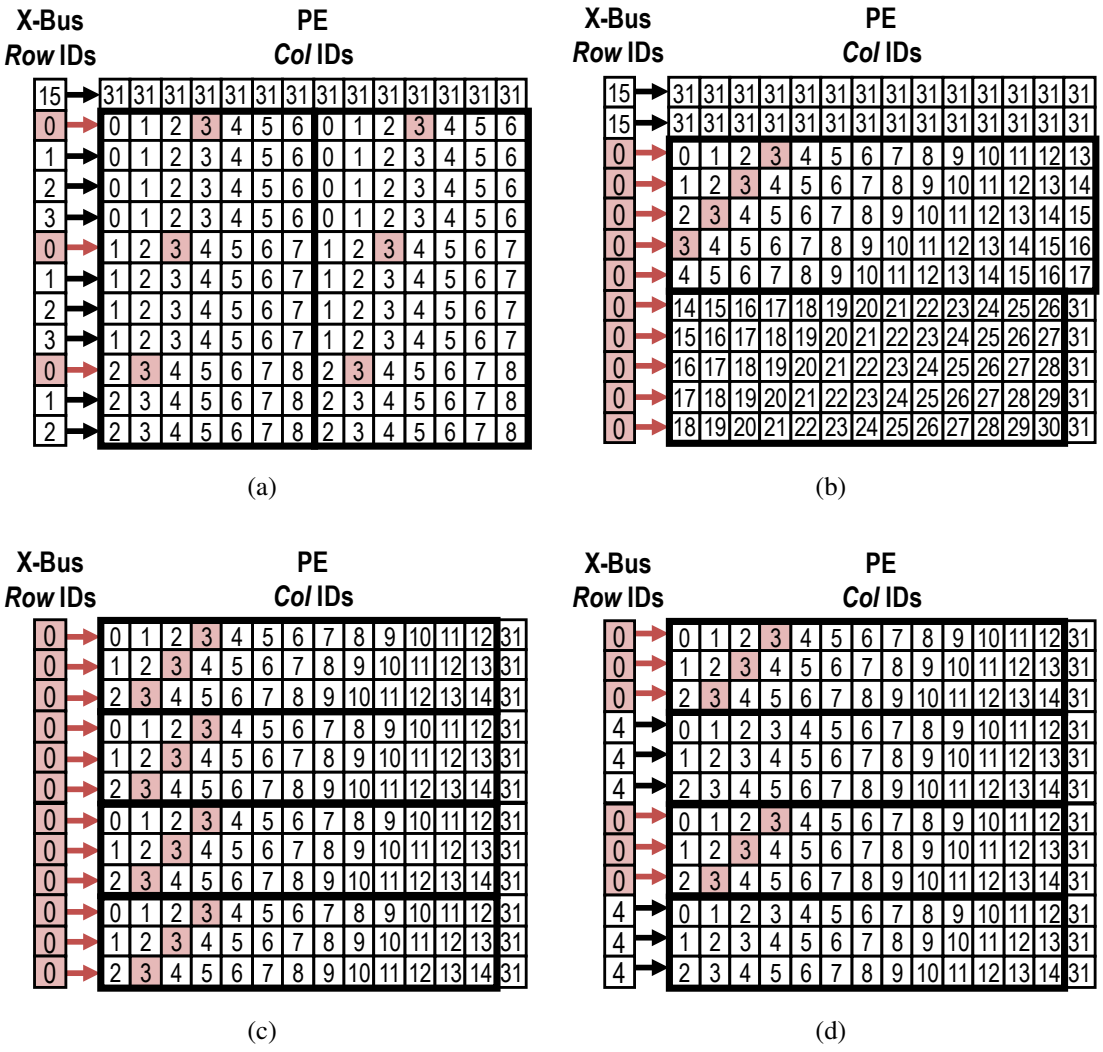


Figure 4-8: The *row* IDs of the X-buses and *col* IDs of the PEs for input activation delivery using GIN in AlexNet layers: (a) CONV1, (b) CONV2, (c) CONV3, (d) CONV4–5. In this example, assuming the tag on the data has $row = 0$ and $col = 3$, the X-buses and PEs in red are the activated buses and PEs to receive the data, respectively.

column, a dedicated 64b data bus is implemented to pass the psums from the bottom PE to the top PE directly. Therefore, a PE can get its input psums either from the psum GIN or LN. The selection is static within a layer, which is controlled by the scan chain configuration bits and only depends on the dataflow mapping of the DNN shape.

4.4.3 Processing Element and Data Gating

Fig. 4-9 shows the architecture of a PE. FIFOs are used at the I/O of each PE to balance the workload between the NoC and the computation. The numbers of filters (p) and channels (q) that the PE processes at once are statically configured into the control of a PE, which determines the state of processing. This configuration controls the pattern with which the PE steps through the three SPads. The datapath is pipelined into 3 stages: one stage for SPad access, and the remaining two for computation. The computation consists of a 16-bit two-stage pipelined multiplier and adder. Since the multiplication results are truncated from 32b to 16b, the selection of 16b out of the 32b is configurable, and can be decided by the dynamic range of a layer from offline experiments. Spads are separated for 3 data types to provide enough accessing bandwidth. The filter SPad is implemented in a 224×16b SRAM due to its large size; the input fmap (ifmap) and psum SPads of size 12×16b and 24×16b, respectively, are implemented using registers.

Data gating logic is implemented to exploit zeros in the input activations for saving processing power. An extra 12b *Zero Buffer* is used to record the position of zeros in the ifmap SPad. If a zero input activation value is detected from the zero buffer, the gating logic will disable the read of the filter SPad and prevent the MAC datapath from switching. Compared with the PE design without the data gating logic, it can reduce the PE power consumption by 45%.

4.5 Implementation Results

The Eyeriss v1 chip shown in Fig. 4-10 was implemented and fabricated in 65-nm CMOS [10] and had been integrated into Caffe [34] (Fig. 4-11). Table 4.3 lists a summary of the chip specifications. At 1.0 V, the peak processing throughput is 33.6 GMAC/sec (GMACS) with a 200 MHz core clock. Also, most of the state-of-the-art DNNs have shapes that lie within the native support of Eyeriss v1. Therefore, they can easily leverage Eyeriss v1 for acceleration with no modification required.

Fig. 4-12a shows the area breakdown of the Eyeriss v1 core, i.e., the area without I/O pads. It includes the logic cells, registers and SRAMs from both the core and link clock

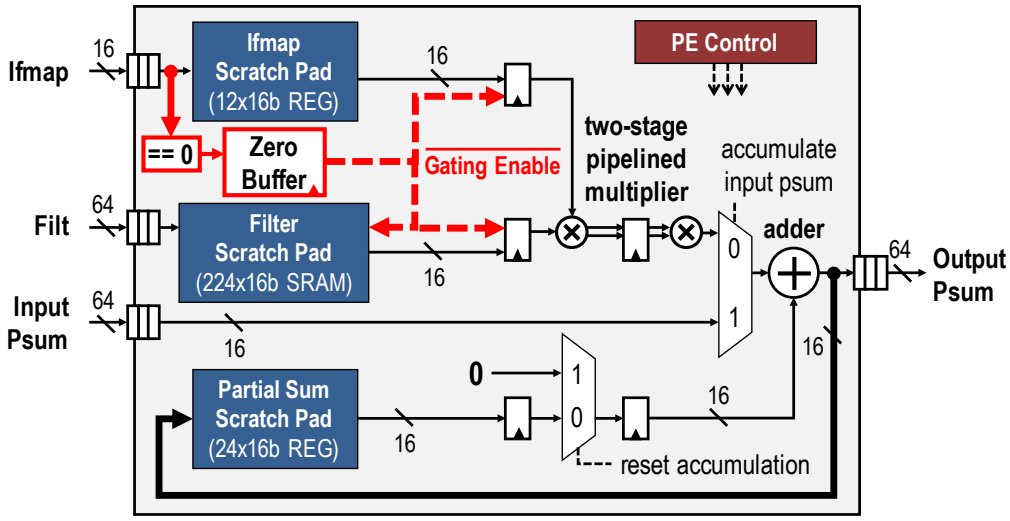


Figure 4-9: The PE architecture. The red blocks on the left shows the data gating logic to skip the processing of zero input activations.

domains. The area of the PE array includes all 168 PEs, and the area breakdown of each PE is shown in Fig. 4-12b. The SPads from all PEs take nearly half of the total area, which is 2.5× larger than that of the GLB. However, the aggregated capacity of the SPads is 1.5× smaller than the size of the GLB. Overall, the on-chip storage, including the GLB and all SPads, takes two-thirds of the total area while the multipliers and adders from all 168 PEs only account for 7.4%.

We benchmark the chip performance using two publicly-available and widely-used DNNs: AlexNet [36] and VGG-16 [61]. The input frames are resized according to the requirement of each DNN: 227×227 for AlexNet and 224×224 for VGG-16. A batch size (N) of 4 and 3 is used for AlexNet and VGG-16, respectively; these batch sizes deliver the highest energy efficiency on Eyeriss v1 according to the optimization in [7].

AlexNet: Table 4.4 shows the measured performance breakdown of the 5 CONV layers in AlexNet at 1.0 V. The chip power consumption gradually decreases through deeper layers, since data gating can leverage more zeros in the input activations. On average, the Eyeriss v1 chip achieves a frame rate of 34.7 fps, or equivalently a processing throughput of 23.1 GMACS. The measured chip power is 278 mW, and the corresponding energy efficiency is 83.1 GMACS/W. The actual throughput is lower than the peak throughput for 3 reasons[1]:

[1]We will formally discuss the cause of these performance degradations in an performance evaluation

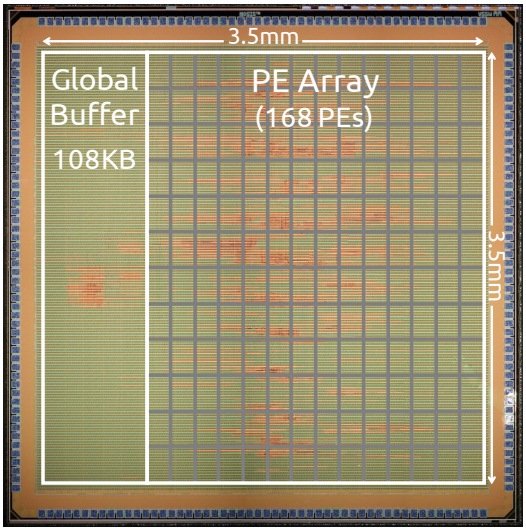


Figure 4-10: Die micrograph and floorplan of the Eyeriss v1 chip.

(1) only 88% of the PEs are active, (2) it takes time to load data from the GLB into the PE array to ramp up each processing pass, and (3) the chip does not perform processing while it is loading input activations from DRAM or dumping output activations to DRAM. The last point, nevertheless, can be optimized with refined control of the DRAM traffic at negligible cost. Therefore, we also provide the *Processing Latency* in Table 4.4 that shows the performance when DRAM traffic is fully overlapped with processing. For a batch of 4 frames, the required DRAM access is 15.4 MB, or 38.4 accesses/pixel.

Fig. 4-13 shows the power breakdown of the chip running CONV1 and CONV5. This is obtained by performing post-place and route simulations using actual workloads as in chip measurement. Different dataflow mappings and data reuse patterns result in different power distributions. Specifically, the power consumed in the SPads as well as multipliers and adders is much lower in CONV5 than CONV1 due to the zeros in input activations. Overall, the ALUs only account for less than 10% of the total power, while data movement related components, including SPads, GLB and NoC, account for up to 45%. This confirms that data movement is more energy consuming that computation. Besides the clock network, the SPads dominate on-chip power consumption, which shows that RS dataflow effectively reuses data locally for reducing DRAM accesses and optimizing overall system energy

framework named Eyexam in Sec 5.2

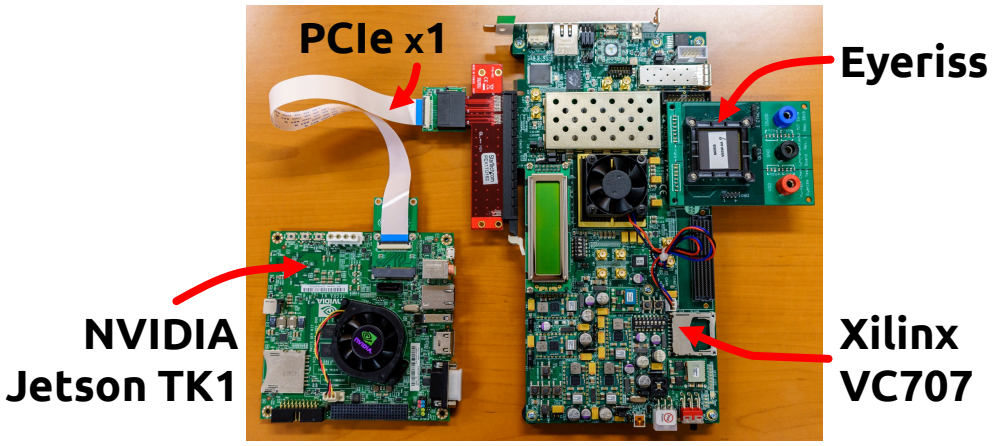


Figure 4-11: The Eyeriss-integrated deep learning system that runs Caffe [34], which is one of the most popular deep learning frameworks. The customized Caffe runs on the NVIDIA Jetson TK1 development board, and offloads the processing of a DNN layer to Eyeriss v1 through the PCIe interface. The Xilinx VC707 serves as the PCIe controller and does not perform any processing. We have demonstrated an 1000-class image classification task [56] using this system, and a live demo can be found in [6].

efficiency as estimated in [7]. This is also why looking at the chip power alone is not sufficient to assess the energy efficiency of the system. Fig. 4-14 shows the impact of voltage scaling on chip performance running AlexNet. The maximum throughput is 45 fps at 1.17 V, and the maximum energy efficiency is 122.8 GMACS/W at 0.82 V.

VGG-16: Table 4.5 shows the measured performance breakdown of the 13 CONV layers in VGG-16 at 1.0 V. On average, the chip operates at 0.7 fps with a measured power consumption of 236 mW. The frame rate is lower than that of AlexNet mainly since VGG-16 requires 23× more computations per frame than AlexNet. The performance, however, depends not only on the computation but also on the shape configuration. For example, CONV1-2 and CONV4-2 both have the same amount of MAC operations, but the former takes nearly 4× longer to process than the latter. This is because the early layers require more processing passes than the deeper layers. Therefore it spends more time on ramping up the processing in the PE array. The large number of processing passes is dictated by the large fmap size. The required DRAM access for a batch of 3 frames is 321.1 MB, or 1038.6 access/pixel.

We have integrated Eyeriss v1 into the processing flow of Caffe [34], which is one of

Technology	TSMC 65nm LP 1P9M
Chip Size	4.0 mm × 4.0 mm
Core Area	3.5 mm × 3.5 mm
Gate Count (logic only)	1176k (2-input NAND)
On-Chip SRAM	181.5K bytes
Number of PEs	168
Global Buffer	108.0K bytes (SRAM)
Scratch Pads (per PE)	filter weights: 448 bytes (SRAM) feature maps: 24 bytes (Registers) partial sums: 48 bytes (Registers)
Supply Voltage	core: 0.82–1.17 V I/O: 1.8 V
Clock Rate	core: 100–250 MHz link: up to 90 MHz
Peak Throughput	16.8–42.0 GMACS
Arithmetic Precision	16-bit fixed-point
Natively Supported DNN Shapes	filter height (R): 1–12 filter width (S): 1–32 num. of filters (M): 1–1024 num. of channels (C): 1–1024 vertical stride: 1, 2, 4 horizontal stride: 1–12

Table 4.3: Chip Specifications

the most popular deep learning frameworks. Fig. 4-11 shows the Eyeriss-integrated deep learning system. The customized Caffe runs on the NVIDIA Jetson TK1 development board, and offloads the processing of a DNN layer to Eyeriss v1 through the PCIe interface. The Xilinx VC707 serves as the PCIe controller and does not perform any processing. We have demonstrated an 1000-class image classification task using this system, and a live demo can be found in [6].

4.6 Conclusions

Eyeriss v1 is a DNN accelerator that supports the row-stationary (RS) dataflow to optimize for energy efficiency and can be reconfigured to support a wide range of DNN shapes and sizes. With a flexible mapping strategy, it can support the mappings of the RS dataflow with high utilization of the PEs, which helps it to achieve high performance. It further

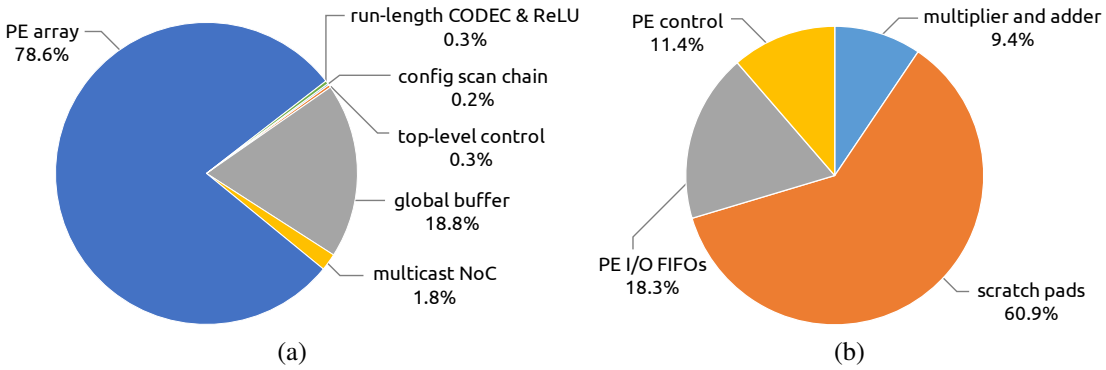


Figure 4-12: Area breakdown of (a) the Eyeriss v1 core and (b) a PE.

Layer	Power (mW)	Total Latency (ms)	Proc. Latency (ms)	Num. of MACs	Num. of Active PEs	Zeros in Ifmaps (%)	Global Buffer Accesses	DRAM Accesses
CONV1	332	20.9	16.5	0.42G	154 (92%)	0.01%	18.5 MB	5.0 MB
CONV2	288	41.9	39.2	0.90G	135 (80%)	38.7%	77.6 MB	4.0 MB
CONV3	266	23.6	21.8	0.60G	156 (93%)	72.5%	50.2 MB	3.0 MB
CONV4	235	18.4	16.0	0.45G	156 (93%)	79.3%	37.4 MB	2.1 MB
CONV5	236	10.5	10.0	0.30G	156 (93%)	77.6%	24.9 MB	1.3 MB
Total	**278**	**115.3**	**103.5**	**2.66G**	**148 (88%)**	**57.53%**	**208.5 MB**	**15.4 MB**

Table 4.4: Performance breakdown of the 5 CONV layers in AlexNet at 1.0 V. Batch size (N) is 4. The core and link clocks run at 200 MHz and 60 MHz, respectively.

exploits data sparsity to reduce PE power by 45% and off-chip data bandwidth by up to 1.9×. Eyeriss v1 is fabricated in a 65nm CMOS, and can process the CONV layers of AlexNet at 35 fps with power consumption at 278 mW. It is 10× more energy efficient than a mobile GPU, and has been integrated in a real-time machine learning system that demonstrates an actual image classification application. Overall, Eyeriss v1 sets a milestone example of a domain-specific processor that serves a wide variety of DNNs in one simple design.

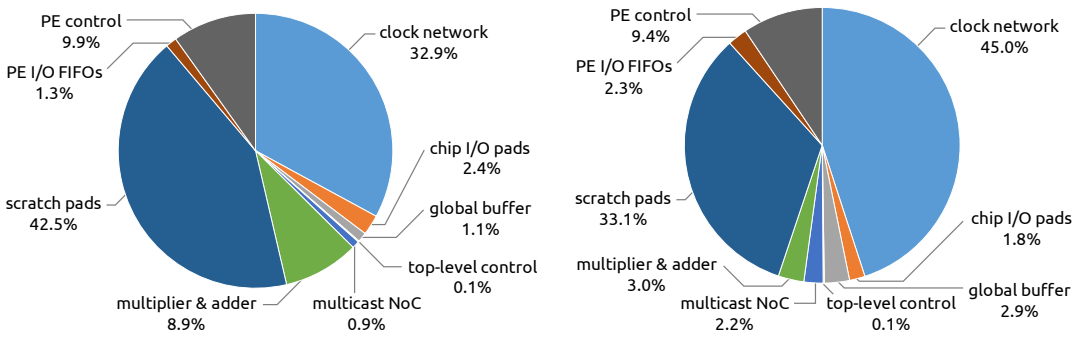


Figure 4-13: Power breakdown of the chip running layer (a) CONV1 and (b) CONV5 of AlexNet.

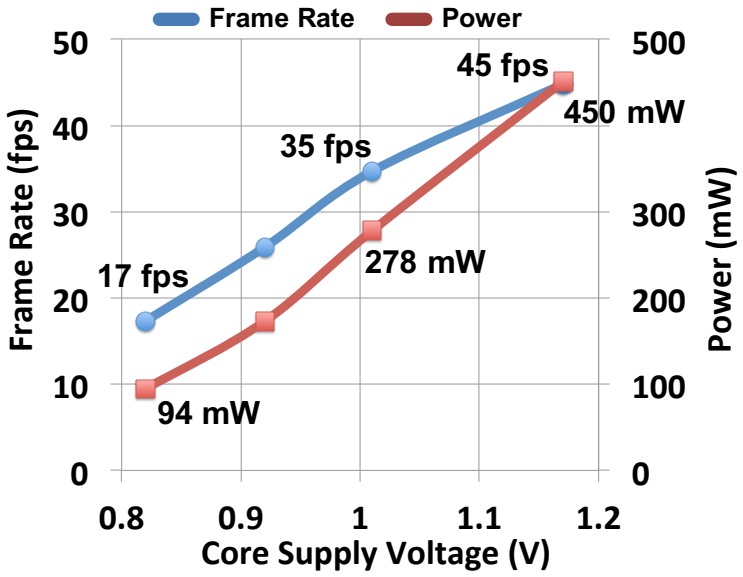


Figure 4-14: Impact of voltage scaling on AlexNet Performance.

Layer	Power (mW)	Total Latency (ms)	Proc. Latency (ms)	Num. of MACs	Num. of Active PEs	Zeros in Ifmaps (%)	Global Buffer Accesses	DRAM Accesses
CONV1-1	247	76.2	38.0	0.26G	156 (93%)	1.6%	112.6 MB	15.4 MB
CONV1-2	218	910.3	810.6	5.55G	156 (93%)	47.7%	2402.8 MB	54.0 MB
CONV2-1	242	470.3	405.3	2.77G	156 (93%)	24.8%	1201.4 MB	33.4 MB
CONV2-2	231	894.3	810.8	5.55G	156 (93%)	38.7%	2402.8 MB	48.5 MB
CONV3-1	254	241.1	204.0	2.77G	156 (93%)	39.7%	607.4 MB	20.2 MB
CONV3-2	235	460.9	408.1	5.55G	156 (93%)	58.1%	1214.8 MB	32.2 MB
CONV3-3	233	457.7	408.1	5.55G	156 (93%)	58.7%	1214.8 MB	30.8 MB
CONV4-1	278	135.8	105.1	2.77G	168 (100%)	64.3%	321.8 MB	17.8 MB
CONV4-2	261	254.8	210.0	5.55G	168 (100%)	74.7%	643.7 MB	28.6 MB
CONV4-3	240	246.3	210.0	5.55G	168 (100%)	85.4%	643.7 MB	22.8 MB
CONV5-1	258	54.3	48.3	1.39G	168 (100%)	79.4%	90.0 MB	6.3 MB
CONV5-2	236	53.7	48.5	1.39G	168 (100%)	87.4%	90.0 MB	5.7 MB
CONV5-3	230	53.7	48.5	1.39G	168 (100%)	88.5%	90.0 MB	5.6 MB
Total	**236**	**4309.5**	**3755.2**	**46.04G**	**158 (94%)**	**58.6%**	**11035.8 MB**	**321.1 MB**

Table 4.5: Performance breakdown of the 13 CONV layers in VGG-16 at 1.0 V. Batch size (N) is 3. The core and link clocks run at 200 MHz and 60 MHz, respectively.

Chapter 5

Highly-Flexible Dataflow and On-Chip Network

5.1 Motivation

The development of DNNs has shown tremendous progress in the past few years. Specifically, there is an increasing focus on reducing the computation complexity of DNNs [64]. This trend is evident in how the iconic DNNs[1] evolve over time. Early models, such as AlexNet [36] and VGG [61], are now considered *large* and over-parameterized. Techniques such as using deeper but narrower network structures and bottleneck layers were therefore proposed to pursue higher accuracy while restricting the size of the DNN (e.g., GoogLeNet [65] and ResNet [28]). This quest further continued with a focus on drastically reducing the amount of computation, specifically the number MACs, and the storage cost, specifically the number of weights. Techniques such as filter decomposition as shown in Fig. 5-1 have since become popular for building *compact* DNNs (e.g., SqueezeNet [32] and MobileNet [30]).

For computer architects, however, this transition brings more challenges than relief due to the change in a key property of DNNs: *data reuse*, which is the number of MACs that each data value is used for (i.e., MACs/data). Fig. 5-2 shows the amount of data reuse for all

[1] We draw examples primarily from the field of computer vision, but this trend is universal across many fields.

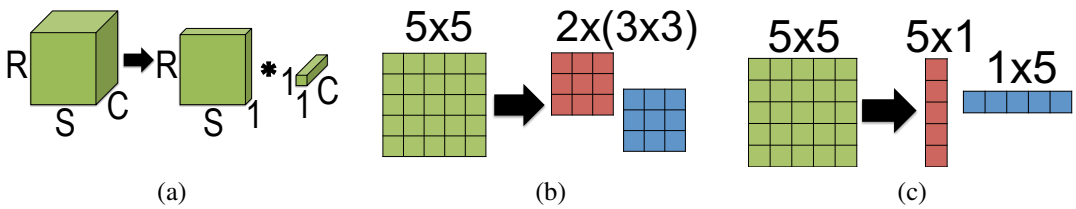
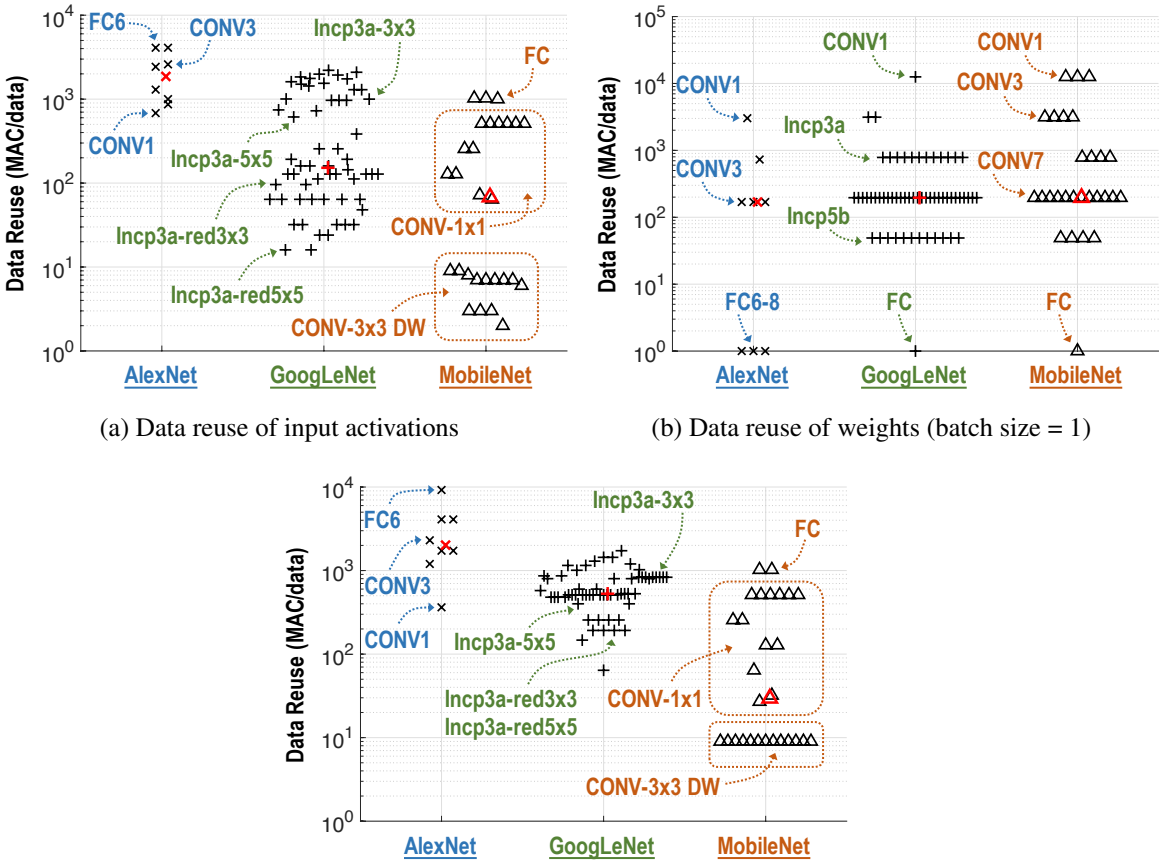


Figure 5-1: Various filter decomposition approaches [30, 61, 66].

three data types (i.e., input activations, weights and psums) in each layer of the three DNNs, ordered from large to compact models: AlexNet, GoogLeNet and MobileNet. When the DNN becomes more compact, the profiled results indicate that the variation in data reuse increases in all data types, and the amount of reuse also decreases in iacts and psums.

This trend makes the design of DNN processors more challenging. For performance, widely varying data reuse is bad in two ways. First, many existing DNN processors [5, 18, 19, 23, 35, 47, 50, 73] depend on a high reuse in certain data dimensions to fully exploit parallelism. For instance, the spatial accumulation array architecture (Figure 5-3a) relies on both input and output channels to map the operations onto the PE array for the spatial reuse of iacts and spatial accumulation of psums. Similarly, the temporal accumulation array architecture (Figure 5-3b) relies on another set of data dimensions to achieve spatial reuse of iacts and weights. When the data reuse in these data dimensions is low, e.g., number of output channels in a layer (M) is less than the height of the PE array, it affects the number of active PEs used in processing (step 4 of Eyexam in Section 5.2). Second, a lower data reuse also implies that a higher data bandwidth from the GLB is required to keep the PEs busy. If the architect designs the data delivery bandwidth for high reuse scenarios, the insufficient data bandwidth can lead to reduced utilization of the active PEs, which further reduces the processor performance (step 6 of Eyexam). However, if the data delivery network is optimized for high bandwidth scenarios, it may not be able to take advantage of data reuse when available (i.e., a low reuse parameter c for the corresponding data type in the energy evaluation framework introduced in Section 3.2). The lack of data reuse makes the optimization of energy efficiency using the optimal mappings less effective.

An additional challenge lies in the fact that all DNNs that the hardware needs to run will *not be known at design time* [12]; as a result, the hardware has to be flexible enough



(a) Data reuse of input activations

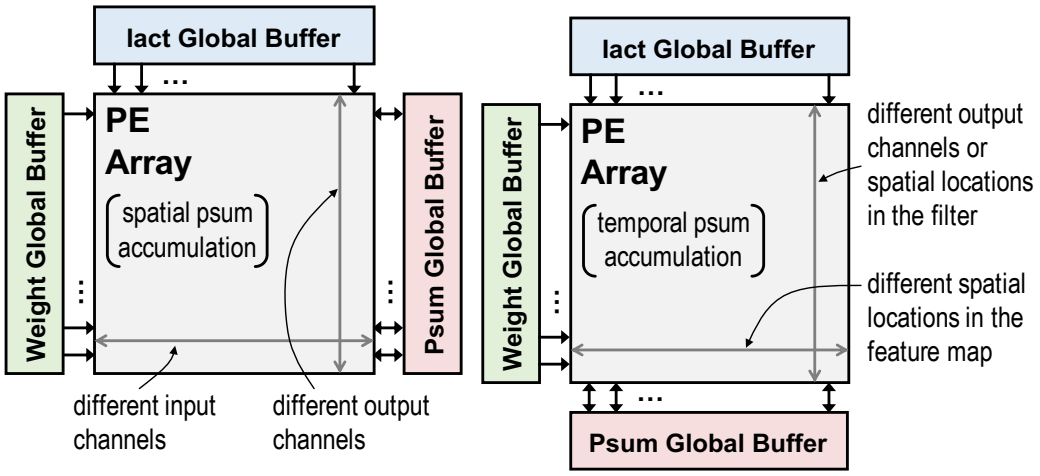
(b) Data reuse of weights (batch size = 1)

(c) Data reuse of partial sums

Figure 5-2: Data reuse in each layer of the three DNNs. Each data point represents a layer, and the red point indicates the median value among the layers in the profiled DNN.

to efficiently support a wide range of DNNs, including *both* large and compact ones. To build a truly flexible DNN processor, the new challenge is to design an architecture that can accommodate a wide range of data reuse among large and compact DNNs. It has to maintain high performance to take advantage of the compact DNNs, but still be able to exploit data reuse with the memory hierarchy and high parallelism when the opportunity presents itself.

In summary, many existing DNN processors suffer from a performance bottleneck when dealing with a wide range of DNNs due to the following reasons: (1) the dataflow relies on certain types of data reuse to achieve high processing parallelism, which can result in low utilization of the parallelism when the amount of data reuse is low; (2) the memory hierarchy for each data type and its corresponding data delivery networks are designed for either (i) high bandwidth and low data reuse or (ii) low bandwidth and high data reuse scenarios



(a) Spatial accumulation array (b) Temporal accumulation array

Figure 5-3: (a) Spatial accumulation array: iacts are reused vertically and psums are accumulated horizontally. (b) Temporal accumulation array: iacts are reused vertically and weights are reused horizontally.

instead of being adaptive to the specific condition of the workload.

Based on these insights, we present the co-design of two key architectural components that are currently the bottleneck for dealing with a more diverse set of DNNs: the dataflow and NoC. Specifically:

- **Flexible Dataflow** (Section 5.3): the new dataflow, named Row-Stationary Plus (RS+), is based on the RS dataflow and further improves on the flexibility by supporting data tiling from all dimensions to fully utilize the PE array, preventing performance loss when the available reuse in certain dimensions is low.

- **Flexible NoC** (Section 5.4): the new NoC, called hierarchical mesh, is designed to adapt to a wide range of bandwidth requirements. When data reuse is low, it can provide high bandwidth from the GLB to keep the PEs busy; when data reuse is high, it can still exploit data reuse to achieve high energy efficiency.

5.2 Eyexam: Framework for Evaluating Performance

In this section, we will present an analysis framework, called Eyexam, that quantitatively identifies the sources of performance loss in DNN processors. Instead of comparing the

overall performance of different designs, which can be affected by many non-architectural factors such as system setup and technology differences, Eyexam provides a step-by-step process that associates a certain amount of performance loss to each architectural design decision (e.g., dataflow, number of PEs, NoC, etc.) as well as the properties of the workload, which for DNNs is dictated by the layer shape and size (e.g., filter shape, feature map size, batch size, etc.).

Eyexam focuses on two main factors that affect performance: (1) the *number of active PEs* due to the mapping as constrained by the dataflow, (2) the *utilization of active PEs*, i.e. percentage of active cycles for the PE, based on whether the NoC has sufficient bandwidth to deliver data to PEs to keep them active. The product of these two components can be used to compute the *utilization of the PE array* as follows

$$\text{utilization of the PE array} = \text{number of active PEs} \times \text{utilization of active PEs}. \quad (5.1)$$

Later in this section, we will see how this approach can use an adapted form of the well-known roofline model [70] for the analysis of DNN processors.

We will perform this analysis on a generic DNN processor architecture based on a spatial architecture that consists of a global buffer and an array of PEs. Each PE can have its own scratchpad (SPad) and control logic, and the PE array communicates with the global buffer through the NoCs. Separate NoCs are used for the three data types, and Fig. 5-4 shows several commonly used NoC designs for different degrees of data reuse and bandwidth requirements. The choice largely depends on how the dataflow exploits spatial data reuse for a specific data type.

The dataflow of a DNN processor is one of the key attributes that define its architecture [8]. In this work, we will feature architectures that support the following four popular dataflows [7, 52]: weight-stationary (WS), output-stationary (OS), input-stationary (IS), and row-stationary (RS).

To help illustrate the capabilities of Eyexam, we will re-examine the simple 1D convolution example in Section 5.2.1, and walk through the key steps of Eyexam in Section 5.2.2 with the 1D convolution. We will then highlight various insights that Eyexam gives on real

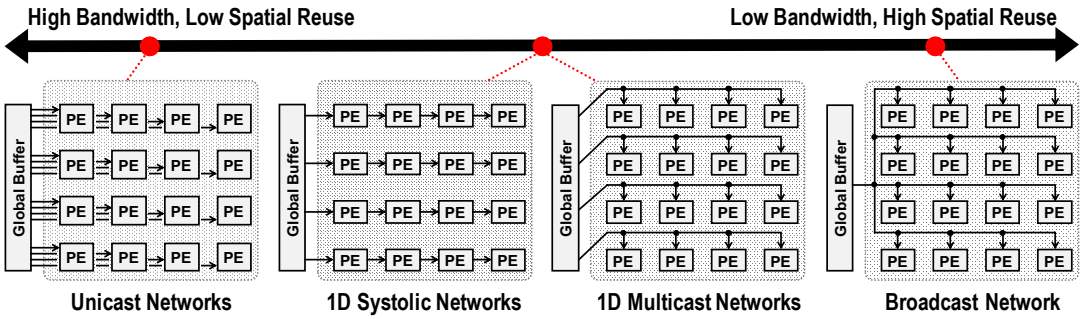


Figure 5-4: Common NoC implementations in DNN processor architectures.

DNN workloads and architectures in Section 5.2.3, which motivates the development of the Eyeriss architecture discussed in Chapter 5.

5.2.1 Simple 1D Convolution Example

We will re-examine the simple 1D convolution example. This example illustrates the two components of the problem. The first is the *workload*, which is represented by the shape of the layer for a 1D convolution. This comprises the filter size R and the input feature map size H and the output feature map size E. The second is the *architecture* of the processing unit, for which a key characteristics is the dataflow shown in Fig. 2-1. In this example, the two `parallel-fors` represent the distribution of computation across multiple PEs (i.e., spatial processing); the inner two `for loops` represent the temporal processing and SPad accesses within a PE, and the outer two `for loops` represent the temporal processing of multiple passes across PE array and GLB accesses. For this example, we assume the input activations and weights fit in the GLB, i.e., the reuse parameter a is 1 for both the input activation and weight.

A mapping assigns specific values to loop limits $E0$, $E1$, $E2$ and $R0$, $R1$, $R2$ to execute a specific workload shape and loop ordering. This assignment of $E0$, $E1$, $E2$ and $R0$, $R1$, $R2$ is constrained by the shape of the workload and the hardware resources. The workload constraints in this example are $E0 \times E1 \times E2 = E$ and $R0 \times R1 \times R2 = R^2$. The architectural constraint in this example is that $E1 \times R1$ must be less than the number of PEs (later we will

[2]We assume perfect factorization in this example. Imperfect factorization will lead to cycles where no work is done.

see that the NoC can pose additional restrictions). The size of the SPad allocated to input activations, psums and weights will restrict $E0$ and $R0$, and the space in the GLB allocated to psums restricts $E1$ and $R1$.

While this is a simple 1D example, it can be extended to additional levels of buffering by adding additional levels of loop nest. Furthermore, extending it to support additional dimensionality (e.g., 2D and channels) will also results in additional loops.

5.2.2 Apply Performance Analysis Framework to 1D Example

The goal of Eyexam is to provide a fine-grain performance profile for an architecture. It is a sequential analysis process that involves seven major steps. The process starts with the assumption that the architecture has infinite processing parallelism, storage capacity and data bandwidth. Therefore, it has infinite performance (as measured in MACs/cycle).

For each of the following steps, certain constraints will be added to reflect changes in the assumptions on the architecture or workload. The associated performance loss can therefore be attributed to that change, and the final performance at one step becomes the upper-bound for the next step.

Step 1 (Layer Shape and Size): In this first step, we look at the impact of the workload constraint, specifically the layer shape (R, H and E), assuming unbounded values for $R1$ and $E1$ since there is no architectural constraints. This allows us to set $R1 = R$, $E1 = E$, and $E2 = E0 = 1$, $R2 = R0 = 1$, so that there is all spatial (i.e., parallel) processing, and no temporal (i.e., serial) processing. Therefore, the performance upper bound is determined by the finite size of the workload (i.e., the number of MACs in the layer, which is $E \times R$).

Step 2 (Dataflow): In this step, we define the dataflow and examine the impact of this architectural constraint. For example, to configure the example loop nest into a weight-stationary (WS) dataflow, we would set $E1 = 1$, $E0 = E$ and $R1 = R$, $R0 = 1$. This means that each PE stores one weight, that weight is reused $E0$ times within that PE, and the number of PE equals the number of weights. This forces the absolute maximum amount of reuse for weights at the PE. The forced serialization of $E0 = E$ reduces the performance upper bound from $E \times R$ to R, which is the maximum parallelism of the dataflow.

Step 3 (Number of PEs): In this step, we define a finite number of PEs, and look at the impact of this architectural constraint. For example, in the 1D WS example, where $E1 = 1$ and $E0 = E$, $R1$ is constrained to be less than or equal to the number of PEs, which dictates the theoretical peak performance. There are two scenarios when the actual performance is less than the peak performance. The first scenario is called spatial mapping fragmentation, in which case R, and therefore $R1$, is smaller than the number of PEs. In this case, some PEs are completely idle throughout the entire period of processing. The second scenario is called temporal mapping fragmentation, in which case R is larger than the number of PEs but not an integer multiple of it. For example, when the number of PEs is 4, $R = 7$ and $R1 = 4$, it takes two cycles to complete the processing, and none of the PEs are completely idle. However, one of the 4 PEs will only be 50% active. Therefore, it still does not achieve the theoretical peak performance. In general, however, if the workload does not map into all of the PEs in all cycles, then some PEs will *not* be used at 100%, which should be taken into account in performance evaluation.

Step 4 (Physical dimensions of the PE array): In this step, we consider the physical dimensions of the PE array (e.g., arranging 12 PEs as 3×4, 2×6 or 4×3, etc.). The spatial partitioning is constrained per dimension which can cause additional performance loss. To explain this step with the simple example, we need to release the WS restriction. Let us assume $E1$ is mapped to the width of the 2D array and $R1$ is mapped to the height of the 2D array. If $E1$ is less than the width of the array or $R1$ is less than the height of the array (spatial mapping fragmentation), not all PEs will be utilized even if without the constraint that $E1 \times R1$ is smaller or equal to the number of PE. A similar case can be constructed for the temporal mapping fragmentation as well. This architectural constraint further reduces the number of active PEs.

Step 5 (Storage Capacity): In this step, we consider the impact of making the buffer storage finite. For example, for the WS dataflow example, if the allocated storage for psums in the GLB is limited, it limits the number of weights that can be processed in parallel, which limits the number of PEs that can operate in parallel. Thus an architectural constraint on how many psums can be stored in the GLB restricts $E1$ and $R1$, which again can reduce the number of active PEs.

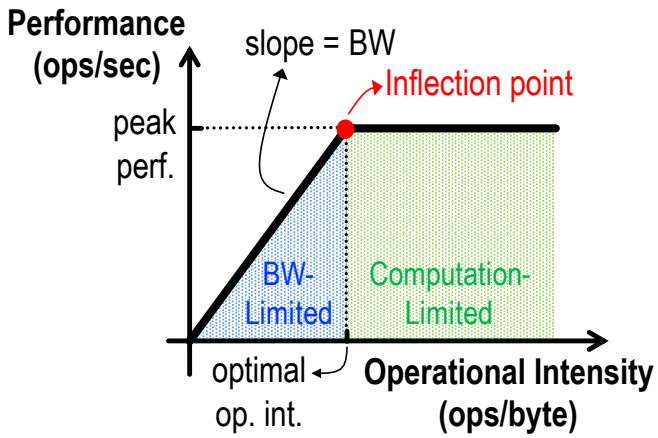


Figure 5-5: The roofline model

Step 6 (Data Bandwidth): In this step, we consider the impact of a finite bandwidth for delivering data across the different levels of the loop nest (i.e., memory hierarchy). The amount of data that needs to be transferred between each level of the loop nest and the bandwidth at which we can transmit the data dictate the speed at which the index of the loop can increment (i.e., number of cycles per MAC). For instance, the bandwidth of the SPad in the PE dictates the increment speed of $r0$ and $e0$, the bandwidth of the NoC and GLB dictates the rate of change of $r1$ and $e1$, and the off-chip bandwidth dictates the rate of change of $r2$ and $e2$. In this work, we will focus on the bandwidth between the GLB and the PEs.

To quantify the impact on performance from insufficient bandwidth, we can adapt the well-known roofline model [70] for the analysis of DNN processors. The roofline model, as shown in Fig. 5-5, is a tool that visualizes the performance of an architecture under various degrees of operational intensity. It assumes a processing core, e.g., PE array, that has insufficient local memory to fit the entire workload, and therefore its performance can be limited by insufficient bandwidth between the core and the memory, e.g., GLB. When the operational intensity is lower than that at the inflection point, the performance will be bandwidth-limited; otherwise, it is computation-limited. The roofline indicates the performance upper-bound, and the performance of actual workloads sit in the area under the roofline.

For this analysis, we adapt the roofline model as follows:

- We use three separate rooflines for the three data types instead of one with the aggregated bandwidth and operational intensity.[3] This helps to identify the performance bottleneck and is also a necessary setup since independent NoCs are used for each data type. However, the performance upper-bound will be the worst case of the three rooflines.

- The roofline is typically drawn with the peak performance of the core and the total bandwidth between the core and memory. However, since we have gone through the first 5 steps in Eyexam, it is possible to get a tighter bound (Fig. 5-6). The leveled part of the roofline is now at the performance bound from step 5; the slanted part of the roofline should only consider the bandwidth to the active PEs for each data type. Since performance is measured in MACs/cycle, the bandwidth should factor in the clock rate differences between processing and data delivery.

- For a workload layer, the operational intensity of a data type is the same as its amount of data reuse in the PE array, including both temporal reuse with the SPad and the spatial reuse across PEs. It is measured in MACs per data value (MAC/data) to normalize the differences in bitwidth.

Step 7 (Varying Data Access Patterns): In this step, we consider the impact of bandwidth varying across time due to the dynamically changing data access patterns (Step 6 only addresses average bandwidth). For the WS example, during ramp up, the weight NoC will require high bandwidth to load the weights into the SPad of the PEs, but in steady state, the bandwidth requirements of the weight NoC will be low since the weights are reused within the PE. The performance upper bound will be affected by ratio of time spent in ramp up versus steady state, and the ratio of the bandwidth demand versus available bandwidth. This step causes the performance point to fall off the roofline as shown in Fig. 5-6. There exist many common solutions to address this issue, including using double buffering or increased bus-width for the NoC. Therefore, we will focus less on the performance loss due to this step in this thesis.

[3] Ideally, we should draw a *roof-manifold* with the operational intensity of each data type on a separate axis; unfortunately, it will be a 4-D plot that cannot be visualized.

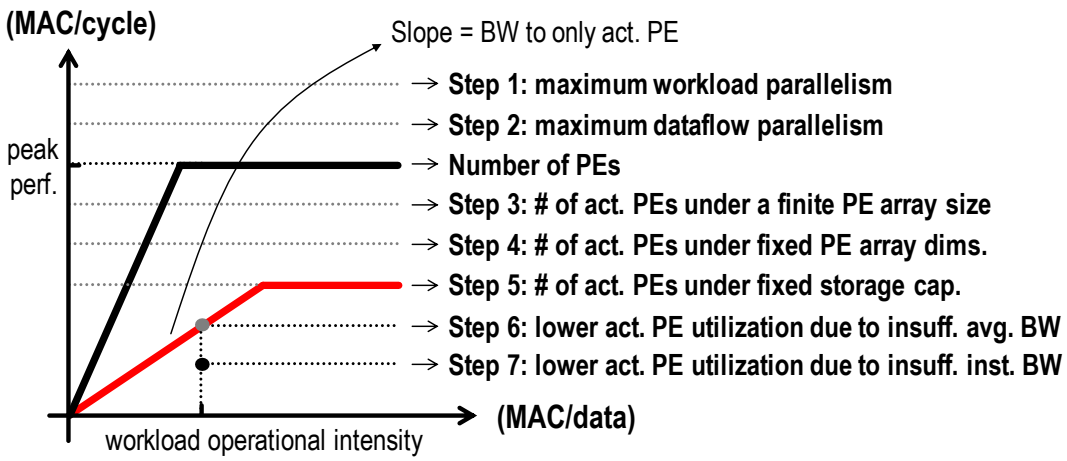


Figure 5-6: Impact of steps on the roofline model.

Table 5.1 summarizes the constraints applied at each step. While Eyexam is useful for examining the impact of each step on performance, it can also be used in the architecture design process to iterate through a design. For instance, if one selects a dataflow in step 2 and discovers that the storage capacity in step 5 is not a good match causing a large performance loss, one could return to step 2 to make a different dataflow design choice and then go through the steps again. Another example is that double buffering could be used in step 7 to hide the high bandwidth during ramp up, however, this would require returning to step 5 to change the effective storage capacity constraints. Eyexam can also be applied to consider the trade-off between performance and energy efficiency in combination with the framework for evaluating energy efficiency as discussed in Section 3.2, as well as consider the impact of sparsity and workload imbalance on performance. However, this is beyond the scope of this thesis.

5.2.3 Performance Analysis Results for DNN Processors and Workloads

In this section, we will highlight some of the observations obtained with Eyexam on DNN processors with real DNN workloads (e.g., AlexNet, MobileNet). We will provide results for architectures from all four representative dataflows, including WS, OS, IS, and Row-Stationary (Chapter 3), with different PE array sizes. The dataflows are evaluated on PE

Step	Constraint	Type	New Performance Bound	Reason for Performance Loss
1	Layer Size and Shape	Workload	Max workload parallelism	Finite workload size
2	Dataflow loop nest	Architectural	Max dataflow parallelism	Restricted dataflow mapping space by defined by loop nest
3	Number of PEs	Architectural	Max PE parallelism	Additional restriction to mapping space due to shape fragmentation
4	Physical dimensions of PEs array	Architectural	Number of active PEs	Additional restriction to mapping space due to shape fragmentation for *each* dimension
5	Fixed Storage Capacity	Architectural	Number of active PEs	Additional restriction to mapping space due to storage of intermediate data (depends on dataflow)
6	Fixed Data Bandwidth	Microarchitectural	Max data bandwidth to active PEs	Insufficient average bandwidth to active PEs
7	Varying Data Access Patterns	Microarchitectural	Actual measured performance	Insufficient instant bandwidth to active PEs

Table 5.1: Summary of steps in Eyexam.

arrays where the height and the width are the same, regardless of the number of PEs.

Fig. 5-7 shows the number of active PEs for the four different architectures in different DNN layers and PE array sizes. It takes into account the mapping of different dataflows in each architecture for different layer shapes under a finite number of PEs. The results are normalized to the total number of PEs in the array. For each bar, the total bar height (white-portion + colored-portion) represent the performance at step 3 of Eyexam, which accounts for the impact of mapping fragmentation due to a finite number of PEs, and the colored-only portion represent the performance at step 4, which further accounts for the impact of the physical dimensions of the PE array. Therefore, the white portion indicates the performance loss from step 3 to 4, which indicates the mapping limitation in the dataflows to adapt to the physical dimensions of the PE array. The results show that

- Fig. 5-7a and 5-7b shows the performance impact when scaling the size of PE array. Many of the architectures are not flexible enough to fully utilize the parallelism when it scales up (i.e., increase number of PEs), which indicates that simply increasing hardware resources is not sufficient to achieve a higher performance.

- Fig. 5-7b and 5-7c shows the performance impact when having to support many different layer shapes. Mapping the different layers onto the same architecture according to its dataflow can result in widely varying performance. For example, the featured IS and OS architectures cannot map well in the layers with smaller feature

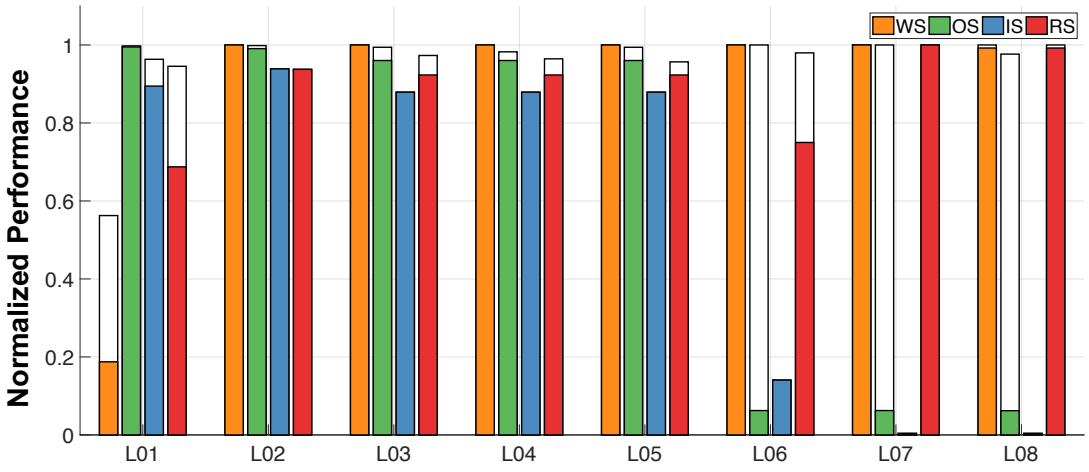
Data Dimension		Common Reasons for Diminishing Dimension (Reuse)
N	batch size	small batch size (usually due to latency requirements)
M	num. of output channels	(1) bottleneck layers, (2) depth-wise layers
C	num. of input channels	(1) layers after bottleneck layers, (2) depth-wise layers, (3) many DNNs with visual inputs only have 3 input channels at the first layer
H / W	input feature map height/width	(1) fmap size reduction in the deep layers of a DNN, (2) fully-connected (FC) layers in CNN/RNN/MLP/etc.
R / S	filter height/width	(1) pointwise layer (i.e., 1×1)
E / F	output feature map height/width	fmap size reduction in the deeper layers of a DNN

Table 5.2: Reason for reduced dimensions of layer shapes.

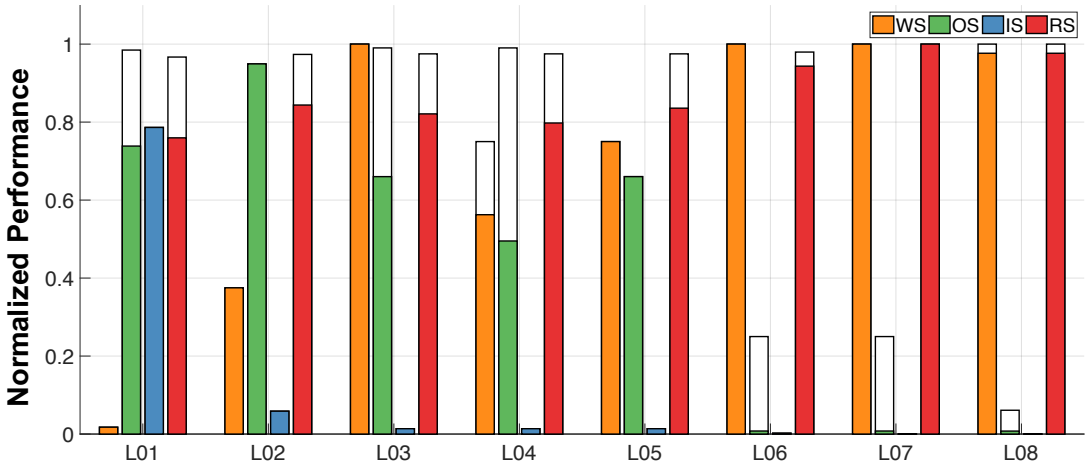
map sizes, while the RS dataflow does not map well in the depth-wise layers due to the lack of channels. Table 5.2 summarizes the common reasons why each data dimension diminishes. In order to support a wide variety of DNNs, the dataflow has to be flexible enough to deal the diminished reuse available in any data dimensions.

- When the PE array size scales up, many of the architectures are not flexible enough to fully utilize the parallelism, which indicates that simply increasing hardware resources is not sufficient to achieve higher performance.

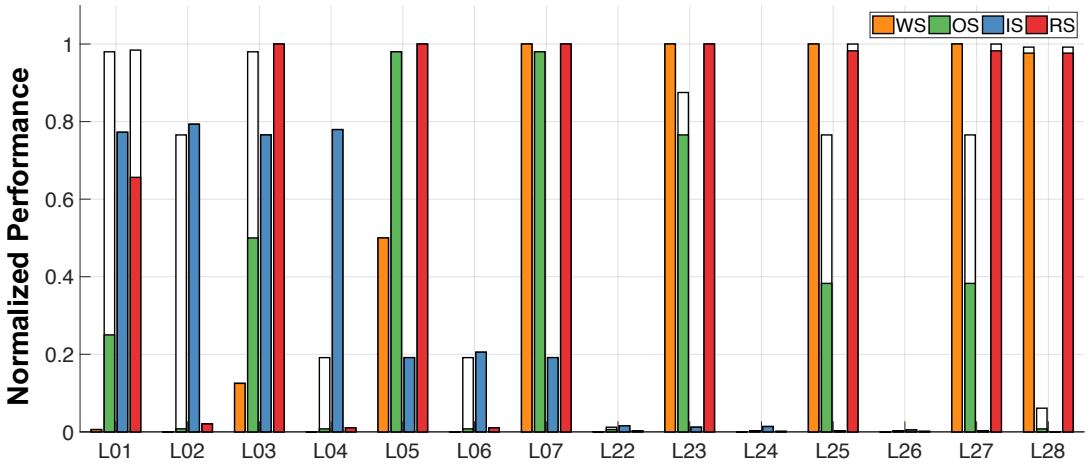
In addition to the loss due to the finite number of PEs and the physical PE array shape, there is loss from insufficient bandwidth for data delivery. To avoid performance loss due to insufficient data bandwidth from the GLB, which results in low utilization of the active PEs (step 6), the NoC design should meet the worst-case bandwidth requirement for every data type. In addition, another NoC design objective is to exploit data reuse to minimize the number of GLB accesses, which is usually realized by the multi-cast of broadcast of data from GLB. On the one hand, for an architecture in which the pattern of spatial data reuse is unchanged with mapping, it is straightforward to meet the two requirements at the same time. For example, if a certain type of data is always reused across an entire PE row or column, the systolic or multicast networks will provide sufficient bandwidth and data reuse from GLB. However, this fixed pattern of data delivery can also cause performance loss in step 3 or 4 of Eyexam. On the other hand, if the architecture support very flexible spatial mappings



(a) AlexNet, 256 PEs



(b) AlexNet, 16384 PEs



(c) MobileNet, 16384 PEs

Figure 5-7: Impact of the number of PEs and the physical dimensions of the PE array on number of active PEs. The y-axis is the performance normalized to the number of PEs.

of operations, which potentially can preserve the performance up to step 5 of Eyexam, the pattern of spatial data reuse can vary widely for different layer shapes. While a single broadcast network can exploit data reuse in any spatial reuse patterns, it sacrifices the data bandwidth from GLB. When the amount of data reuse is low, e.g., delivering weights in FC layers with a small batch size, the broadcast network will result in significant performance loss. Therefore, step 6 will become a performance bottleneck. We will address this problem with a proposed flexible architecture in Section 5.3 and 5.4.

5.3 Flexible Dataflow: Row-Stationary Plus (RS+)

The RS dataflow was designed with the goal to optimize for the best overall system energy efficiency. While it is already more flexible than other existing dataflows as shown in Chapter 3, we have identified several techniques that can be applied to further improve the flexibility of the RS dataflow to deal with a more diverse set of DNNs.

The improved dataflow, named Row-Stationary Plus (RS+), is defined in Fig. 5-8. *The computation of each row convolution is still stationary in the SPad of each PE*, as $f0$ and $s0$ always loop through the entire dimension of F and S, respectively. The key features of the RS+ are summarized as follows:

- Additional loops $g1$ and $n1$ are added at the NoC level compared to the RS dataflow, which provides more options to parallelize different dimensions of data for processing. In fact, since row-stationary always fully loops through data dimensions F and S at the SPad level in loops $f0$ and $s0$, respectively, all the rest of the data dimensions are provided at the NoC level as options for parallel processing. As an example, one type of layer that benefits the most from this is the DW CONV layer [30], in which the number of input and output channels are both one (i.e., $C = M = 1$). The RS dataflow cannot fully utilize the PE array due to the lack of channels, while the RS+ dataflow adapts better as shown in Fig. 5-9.

- Not only does the RS+ dataflow allow the parallelization of processing from more data dimensions, it allows data to be tiled from multiple dimensions and mapped in parallel

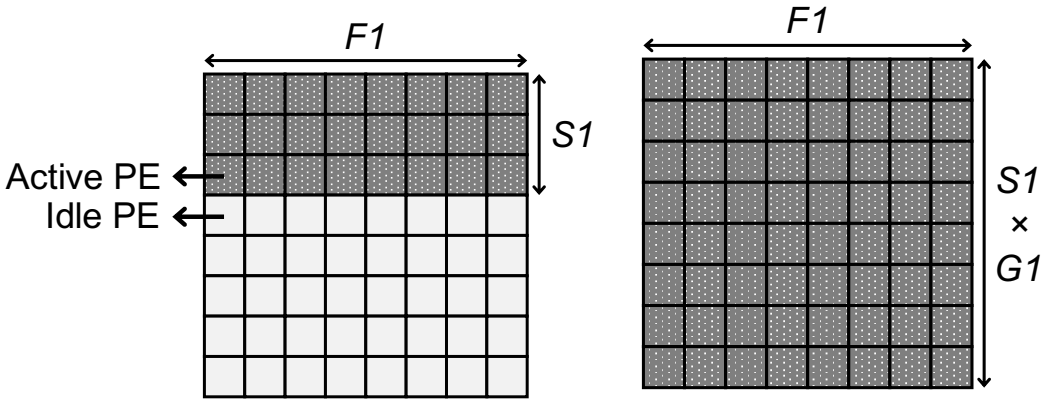
```
Input Fmaps:    I[G][N][C][H][W]
Filter Weights: W[G][M][C][R][S]
Output Fmaps:   O[G][N][M][E][F]

// DRAM level loops
for (g3=0; g3<G3; g3++) {
  for (n3=0; n3<N3; n3++) {
    for (m3=0; m3<M3; m3++) {
      for (e3=0; e3<E3; e3++) {
        // Global buffer level loops
        for (n2=0; n2<N2; n2++) {
          for (e2=0; e2<E2; e2++) {
            for (c2=0; c2<C2; c2++) {
              for (r2=0; r2<R2; r2++) {
                for (m2=0; m2<M2; m2++) {
                  // NoC level loops
                  parallel-for (g1=0; g1<G1; g1++) {
                    parallel-for (n1=0; n1<N1; n1++) {
                      parallel-for (m1=0; m1<M1; m1++) {
                        parallel-for (e1=0; e1<E1; e1++) {
                          parallel-for (c1=0; c1<C1; c1++) {
                            parallel-for (r1=0; r1<R1; r1++) {
                              // SPad level loops
                              for (e0=0; e0<E0; e0++) {
                                for (n0=0; n0<N0; n0++) {
                                  for (f0=0; f0<F; f0++)  {
                                    for (s0=0; s0<S; s0++)  {
                                      for (c0=0; c0<C0; c0++) {
                                        for (m0=0; m0<M0; m0++) {
                                          O += I × W;
}}}}}}}}}}}}}}}}}}}}
```

Figure 5-8: The definition of the Row-Stationary Plus (RS+) dataflow. For simplicity, we ignore the bias term and the indexing in the data arrays.

onto the same PE array dimension (i.e., height or width) at the same time, which is similar to the idea of mapping replication as introduced in Eyeriss v1 (Chapter 4). For example, an PE array with height of 16 can be fully utilized by mapping both $C1 = 4$ and $M1 = 4$ simultaneously onto the height of the array.

- The data tile from the same dimension can also be mapped spatially onto different physical dimensions. For example, a $M1$ of 16 can be split into $M1_{horz} = 4$ and $M1_{vert} = 4$, which are the portions of $M1$ that are mapped horizontally and vertically on the PE array, respectively. This creates more flexibility for the mapper to find a way to fully utilize the PE array.



(a) DW CONV: RS Mapping (b) DW CONV: RS+ Mapping

Figure 5-9: The mapping of depth-wise (DW) CONV layers [30] with the (a) RS and (b) RS+ dataflows.

- In the RS dataflow, all rows of the filter are mapped spatially in order to exploit the available data reuse in the 2D convolution. However, this can create spatial mapping fragmentation if the PE array height is not an integer multiple of the filter height, and results in lower utilization of the PE array. In the RS+ dataflow, this restriction is relaxed by allowing tiling in data dimension R with loop $r1$. When the mapping is optimized for performance, the mapper can find R1 that best fits in the PE array height. However, when the mapping is optimized for energy efficiency, the mapper can still find the same mapping as in the RS dataflow by setting $R1 = R$.

- An additional loop $e0$ is added at the SPad level comparing to the RS dataflow, which concatenates $E0$ fmap rows to be computed with the same row of weights in a PE, therefore creating more reuse of weights in the SPad. However, this adds a storage requirement for the additional fmap rows to be stored in the global buffer. Therefore, $E0$ is constrained by the size of the global buffer.

In summary, the RS+ dataflow provides a much higher flexibility in mapping than the RS dataflow. In fact, the mapping space of the RS+ dataflow is a strict super-set of the RS dataflow, i.e., the optimal mapping of the RS+ dataflow for a DNN layer with any optimization objective is at least the same or better than the optimal mapping of the RS dataflow. With a more powerful dataflow, the remaining challenge is to deliver data to the

PEs according to the mapping for processing. In the next section, we will describe a new NoC that can unleash the full potential of the RS+ dataflow.

5.4 Flexible Network: Hierarchical Mesh NoC

NoC design is a well-studied field in various contexts, e.g., manycore processors [33]. However, the complexity of a core in a multicore processor is much higher than that of a PE, which usually has specialized datapaths with little control logic, and the memory hierarchy is also highly customized; thus, a DNN processor cannot use the conventional sophisticated NoC used to connect cores on multicore processor [37]. Accordingly, most DNN processors adopt a NoC implementation with minimum routing and flow control complexity, such as the ones shown in Fig. 5-4, and it is important to keep it that way to maintain the efficiency of the architecture.

However, these NoC implementations have their drawbacks as shown in Fig. 5-10. On the one hand, the broadcast network can achieve high spatial data reuse. More importantly, it allows any patterns of data reuse, making it possible to be applied for any dataflow. However, the data bandwidth from the source (e.g., the global buffer) is quite limited. If the amount of data reuse is low, i.e., different destinations (e.g., PEs) require unique data for processing, the broadcast NoC has to deliver data to different destinations sequentially, resulting in reduced performance. Even though it is possible to increase the data bus width to deliver more data to the same destination at once, it would create high buffering requirements at each destination. This solution also does not scale as the buffering requirement will go higher with the number of destinations. On the other hand, the unicast network can achieve high bandwidth from the source by leveraging many independent sources, e.g., banked memory. However, it cannot exploit spatial data reuse, which will reduce the energy efficiency. A possible solution is the all-to-all network, which has the ultimate flexibility that can adapt between delivering high spatial data reuse and high data bandwidth from the source. However, it is very hard to scale as both the implementation cost and energy consumption will increase quadratically with the number of sources/destinations.

In order to build a NoC that is both adaptive and easy to scale, it is important to first

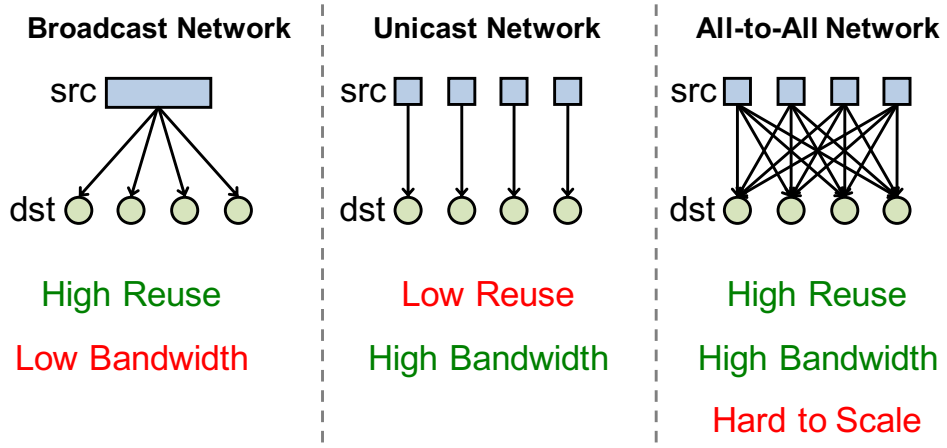


Figure 5-10: The pros and cons of different NoC implementations.

understand the specific requirements from the dataflow, which is RS+ in this case. From the features described in Section 5.3, we can summarize the types of data delivery patterns required by the RS+ dataflow in the examples shown in Fig. 5-11. These three examples show four data delivery patterns:

- Broadcast: weights in Example 1 and inputs in Example 3
- Unicast: inputs in Example 1 and weights in Example 3
- Grouped Multicast: weights in Example 2
- Interleaved Multicast: inputs in Example 2

Note that the patterns always come in a pair, and there are two possible combinations. The broadcast-unicast pair is required when the data dimensions mapped onto the same physical dimension of the PE array only address one data type. For example, when only the output channel (M) dimension is mapped onto the entire PE array width, the pattern will be the same as in Example 3, where the same input activation is broadcast across the PEs to be paired with unique weights from different output channels. The multicast pair is required when multiple data dimensions are mapped onto the same physical PE array dimension, and each data type is addressed by a unique subset of these data dimensions. For example, when the input fmap height (E) and output channels (M) are mapped simultaneously onto the

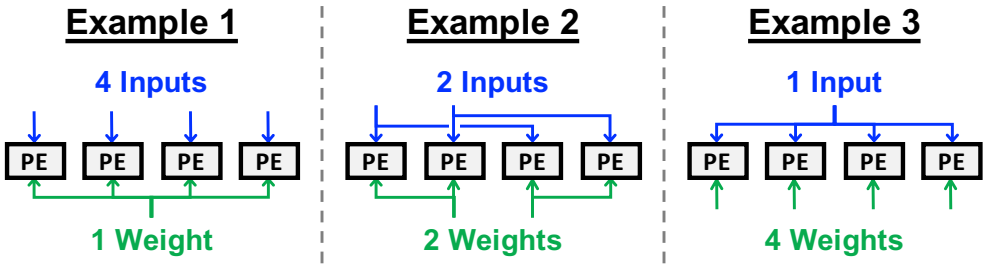


Figure 5-11: Example data delivery patterns of the RS+ dataflow.

width of the PE array, the pattern will be similar to the one in Example 2. In order to support a wide range of mappings, it is critical to be able to support all four data delivery patterns.

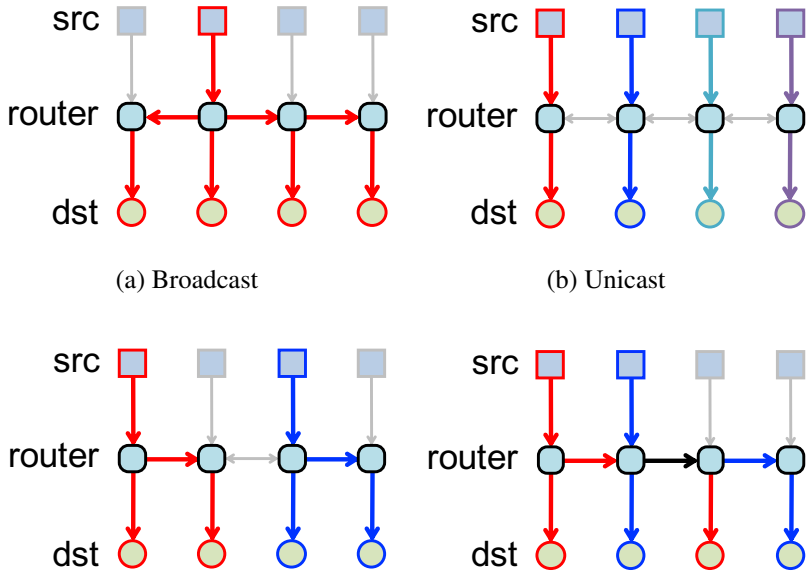
One possible solution that has the potential to support these data delivery patterns and is easy to scale is the mesh network. The mesh network can be constructed by taking the unicast network, inserting a router in between each pair of source and destination, and linearly connect the routers. Fig. 5-12 demonstrates how the mesh network can be configured to support each of the data delivery patterns. While it can easily support broadcast, unicast and grouped multicast, the interleaved multicast will cause a problem as the middle route between the routers (colored in black) needs a higher bandwidth than other routes. The number of routes with higher bandwidth requirement and the required bandwidth itself also grow with the size of the mesh network. Therefore, the mesh network alone is still not the answer.

To solve this problem, we propose a new NoC based on the mesh network, called a hierarchical mesh network. Fig. 5-13 shows a simple example of a 1D hierarchical mesh. It has the following features:

- The architectural components, including sources, destinations and routers, are grouped into clusters. The size of each type of cluster is determined at design time and fixed at compile time and runtime.

- The router clusters are connected linearly as in a mesh network. The individual routers in between adjacent clusters are one-to-one connected.

- Each router cluster is connected with a source cluster with one-to-one links between



(a) Broadcast (b) Unicast

(c) Grouped Multicast (d) Interleaved Multicast

Figure 5-12: Configurations of the mesh network to support the four different data delivery patterns.

each pair of source and router.

- Each router cluster is also connected with a destination cluster. The links between the routers and destinations in the cluster are all-to-all connections.

Fig. 5-14 shows how the hierarchical mesh network supports the four data delivery patterns. It is able to support all four patterns by explicitly defining the bandwidth between all types of clusters through setting the size of these clusters at design time. Compared with the plain mesh network, only the all-to-all network in between the router cluster and the destination cluster incurs a higher cost, and this cost can be well-controlled locally. When setting the size of these clusters, the key characteristics to consider are (1) what is the bandwidth required from the source cluster in the worst case, i.e., unicast mode, (2) what is the bandwidth required in between the router clusters in the worst case, i.e., interleaved multicast mode, and (3) what is the tolerable cost of the all-to-all network.

The hierarchical mesh network has two advantages. First, there is no routing required at runtime. All active routes are determined at configuration time based on the specific data delivery pattern in use. As a result, no flow control is required, and the routers are simply

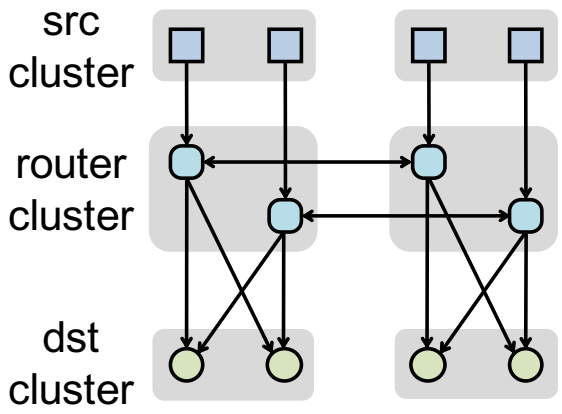
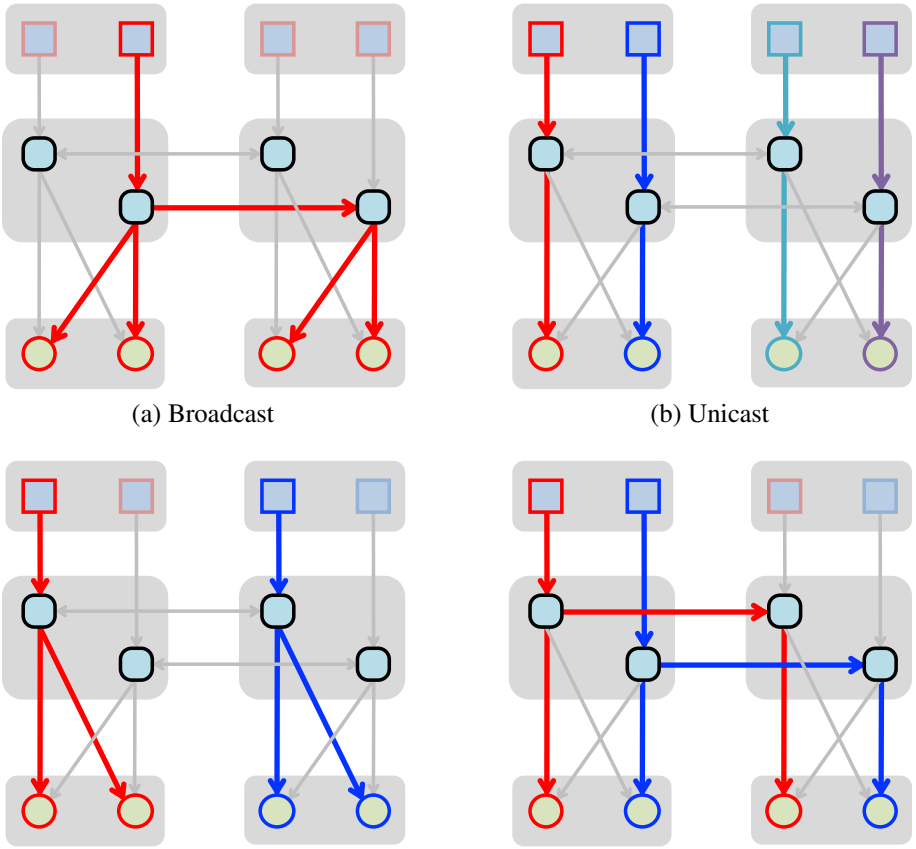


Figure 5-13: A simple example of a 1D hierarchical mesh network.

multiplexers for circuit-switched routing that has minimum implementation cost. Second, the network can be easily scaled. Once the cluster size is determined, the entire architecture can be scaled at the cluster level, in which case the cost only increases linearly instead of quadratically as in the plain all-to-all network.

One restriction the hierarchical mesh network imposes on the mapping space of the RS+ dataflow is that the tile size of the mapped data dimensions is further constrained by (1) the cluster size and (2) the number of clusters in addition to the total height and width of the PE array. For example, in the multicast delivery patterns, the data delivered with grouped multicast has the maximum reuse constrained by the size of the PE cluster. Similarly, the data delivered with interleaved multicast has the maximum reuse constrained by the number of clusters. In Section 5.5.2, we will discuss how does this affect the performance of the architecture.

Fig. 5-15 shows an example DNN accelerator built based on the hierarchical mesh network. The router clusters are now connected in a 2D mesh. The global buffer is banked and distributed into each source cluster, and the PEs are grouped into the destination clusters instead of one single array.



(a) Broadcast (b) Unicast

(c) Grouped Multicast (d) Interleaved Multicast

Figure 5-14: Configurations of the hierarchical mesh network to support the four different data delivery patterns.

5.5 Performance Profiling

In this section, we profile the performance of the co-design of the RS+ dataflow and the hierarchical mesh network. We will first describe the methodology used to conduct the experiment in Section 5.5.1, and then demonstrate and discuss the experiment results in Section 5.5.2.

5.5.1 Experiment Methodology

The performance results are based on the number of active PEs and their utilization in cases where the utilization of the active PEs are not 100% due to the limited bandwidth. The number of active PEs for the various steps in Eyexam are evaluated analytically. For situations where the performance is limited by characteristics of the mappings, we

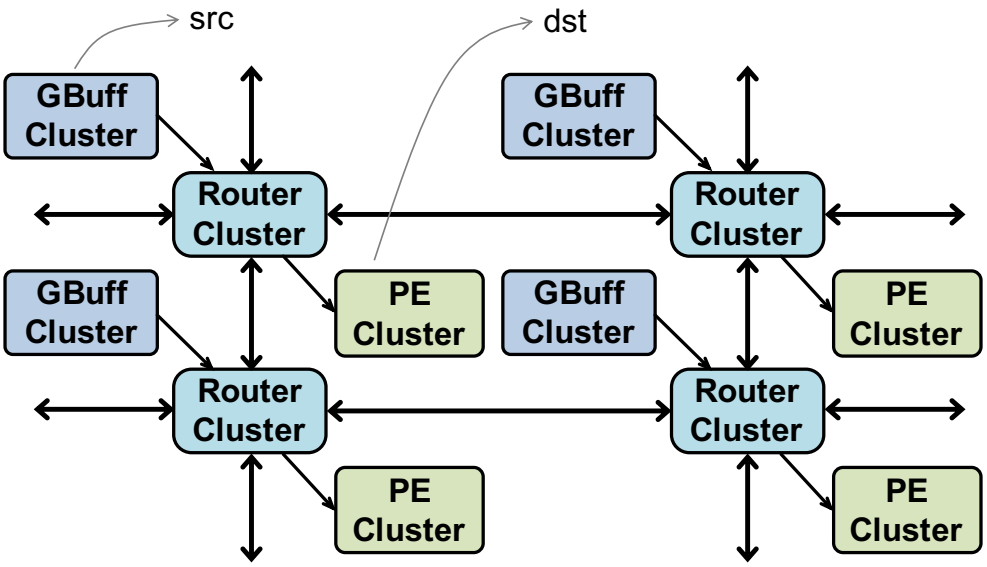


Figure 5-15: A DNN accelerator architecture built based on the hierarchical mesh network.

exhaustively search the mapping space and analytically model the consequences of each mapping.

For the cases with performance limited by bandwidth constraints, we combine the Eyexam results with roofline models for each of the data types. The bandwidths of the rooflines are determined with an analytical model of the hierarchical mesh NoC.

5.5.2 Experiment Results

In this section, we will examine the performance of the combination of the RS+ dataflow and the hierarchical mesh network, named Eyeriss v1.5, and compare it to the combination of the RS dataflow and the broadcast network implemented in Eyeriss v1 [11][4]. First, we will compare them in terms of the number of active PEs from the optimal mappings generated by their respective dataflows. Then, we will discuss the impact of data bandwidth on the performance.

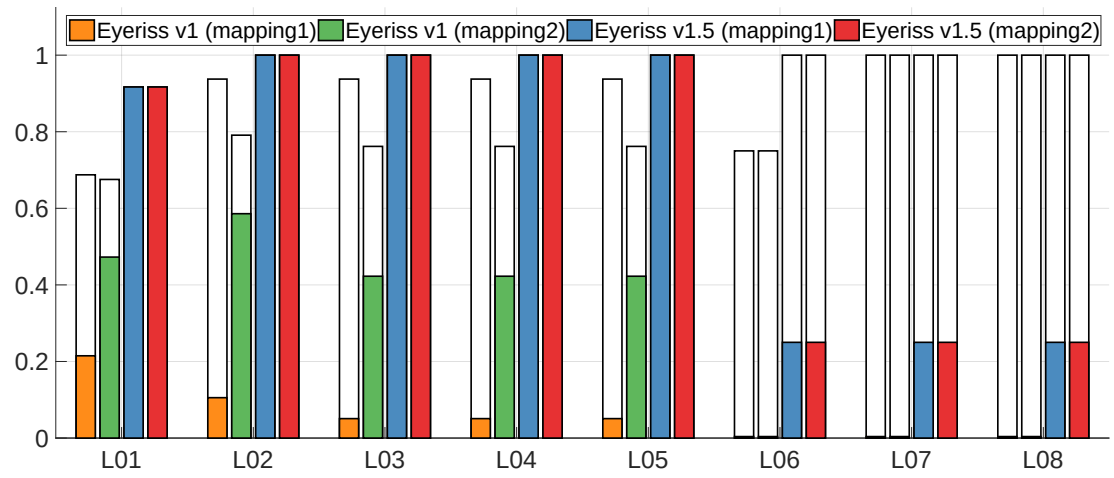
For each architecture, we simulate with three different PE array sizes, including 256, 1024 and 16384 PEs. For Eyeriss v1, the PE array is a square, i.e., 16×16, 32×32 and

[4]The NoC in Eyeriss v1 is a broadcast network that is implemented with multicast capabilities.

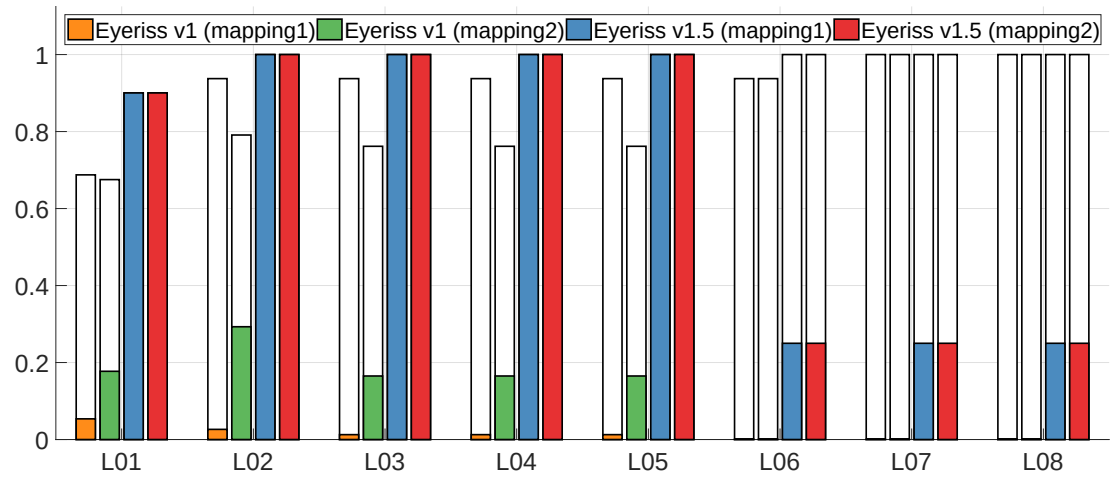
128×128. For Eyeriss v1.5, we fix the PE cluster size at 4×4, and scale the number of PE clusters at 4×4, 8×8 and 32×32. In terms of data bandwidth, Eyeriss v1 uses a single broadcast network for each of the three data types, which is capable of delivering 1 data/cycle. For Eyeriss v1.5, each data type has a separate hierarchical mesh network with a router cluster size of 4. Each pair of source to router link can deliver 1 data/cycle. We assume a PE architecture similar to the one described in [11], which has the SPad sizes for input activation, weight and psum at 12, 192 and 16, respectively. The PE is also assumed to be able to sustain a processing throughput of 1 MAC/cycle if not bandwidth limited.

We evaluated the architectures with three different DNNs, including AlexNet [36], GoogLeNet [65] and MobileNet [30]. Each layer in the three DNNs is mapped on the two architectures for processing independently, and two types of performance are quantified: (1) *the number of active PEs*, which is the performance at step 5 of Eyexam assuming an infinite data bandwidth. (2) *the overall utilization of the PE array*, which further models the impact of a finite bandwidth on the performance and is the performance at step 6 of Eyexam. We serialize the layers in each DNN and name them starting from L01. The layers in the inception modules of GoogLeNet are serialized in the following order: 3×3 reduction, 5×5 reduction, 1×1 CONV, 3×3 CONV, 5×5 CONV and 1×1 CONV after pooling. In all cases we use a batch size of 1, which is a crucial criterion for many low-latency applications but also greatly reduces the reuse of weights.

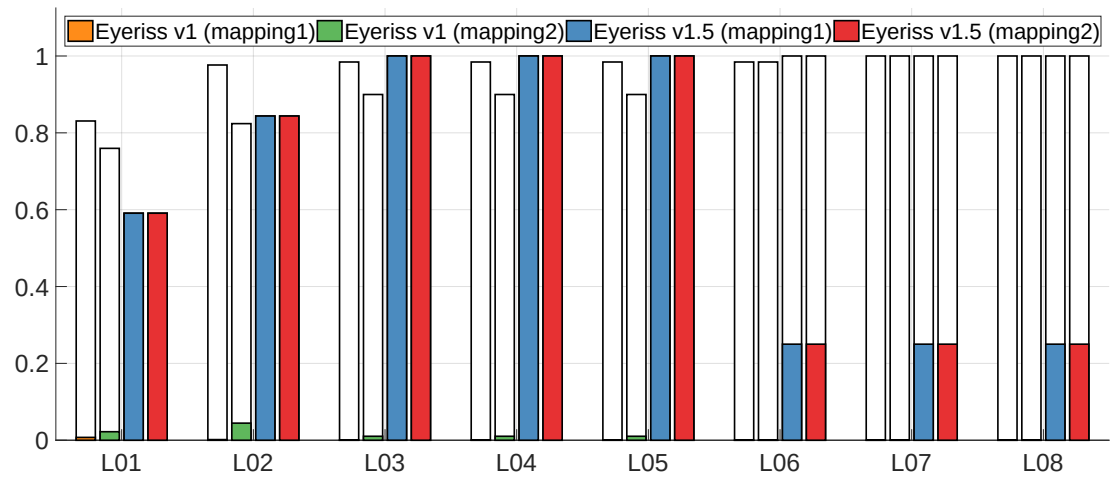
Fig. 5-16, 5-17 and 5-18 shows the performance comparison between Eyeriss v1 and Eyeriss v1.5 for each DNN layer in AlexNet, GoogLeNet and MobileNet, respectively, at different PE array sizes. For each PE array size, the performance in the y-axis is normalized to its total number of PEs (i.e., peak performance). The total bar height (white + colored portion) indicates the performance accounting for the impact of workload and architectural constraints excluding the impact of the NoC bandwidth, i.e., it is the performance assuming all active PEs run at 100% utilization (step 5 of Eyexam). The color-only portion of the bars indicates the performance further accounting for the impact of the finite bandwidth from the specific NoC design, which reduces the utilization of the active PEs and represents the overall utilization of the PE array (step 6 of Eyexam). Therefore, the white portion of the bars, if any, indicates the performance loss due to the constraint of a finite NoC bandwidth.



(a) 256 PEs

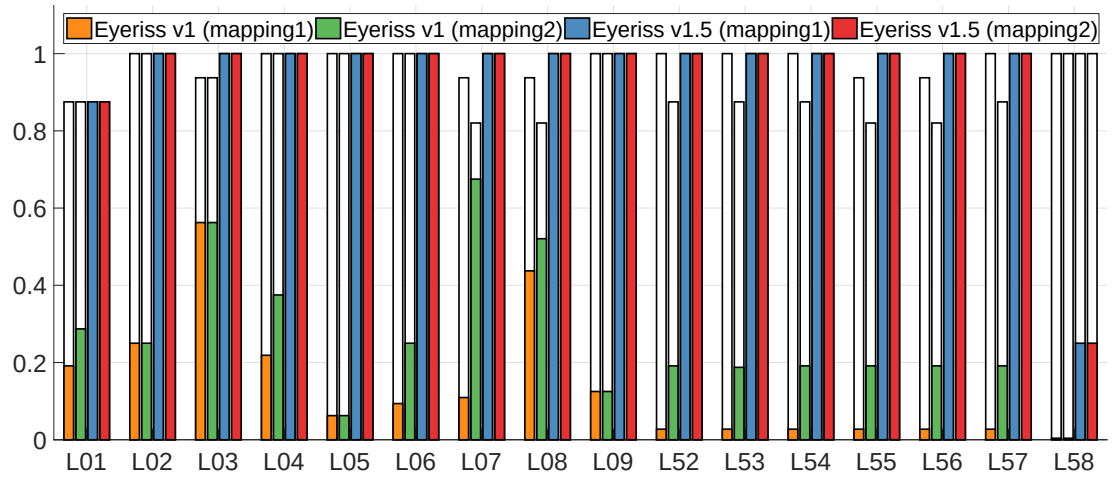


(b) 1024 PEs

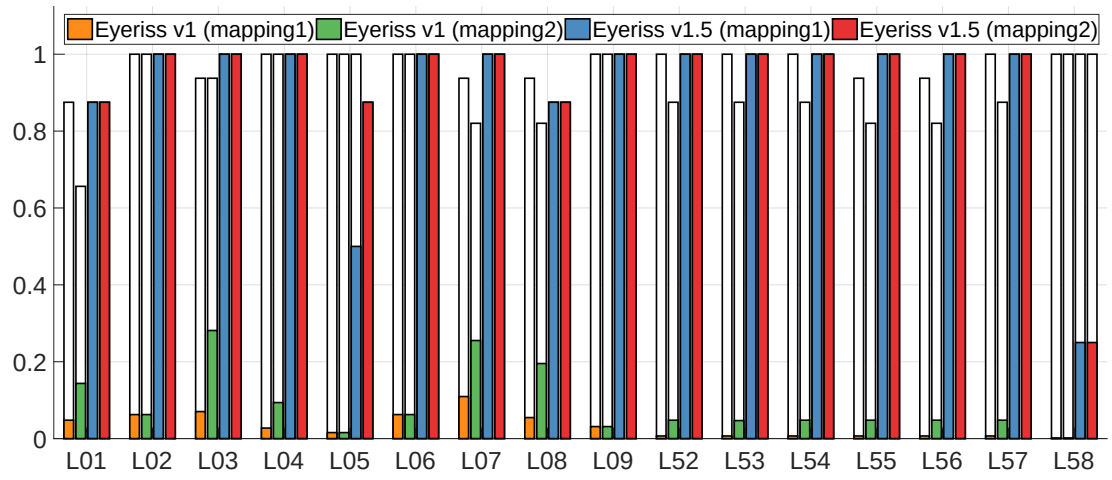


(c) 16384 PEs

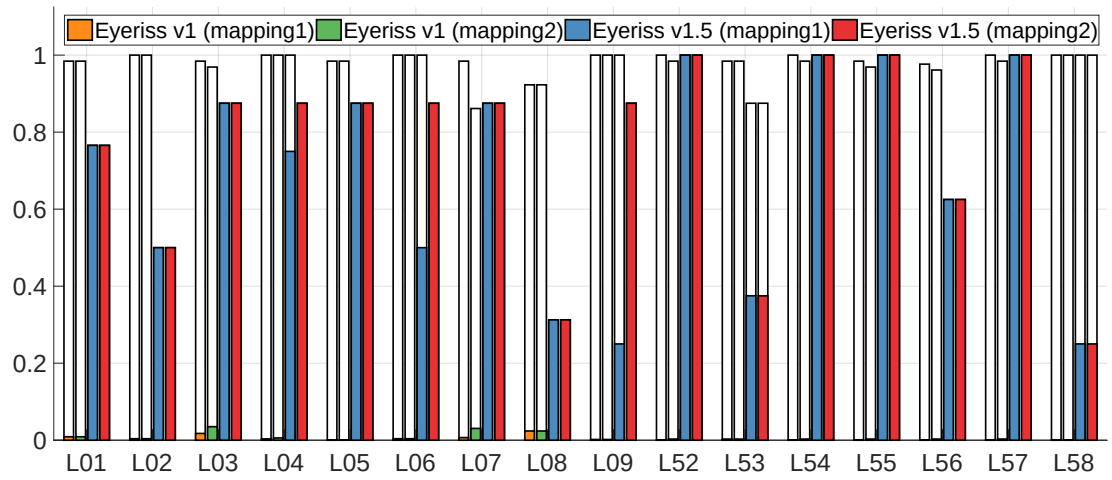
Figure 5-16: Performance of AlexNet at PE array size of (a) 256, (b) 1024 and (c) 16384. Performance is normalized to the peak performance.



(a) 256 PEs

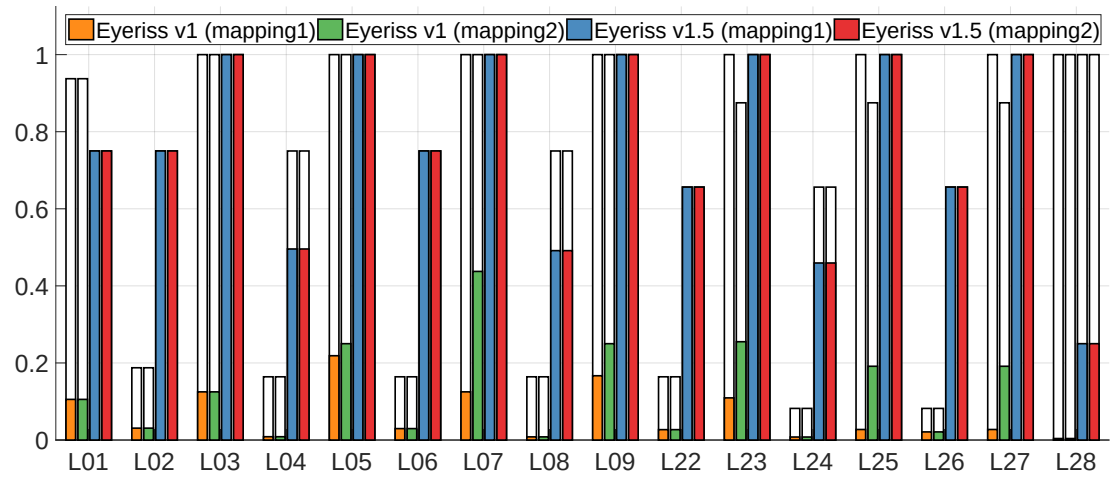


(b) 1024 PEs

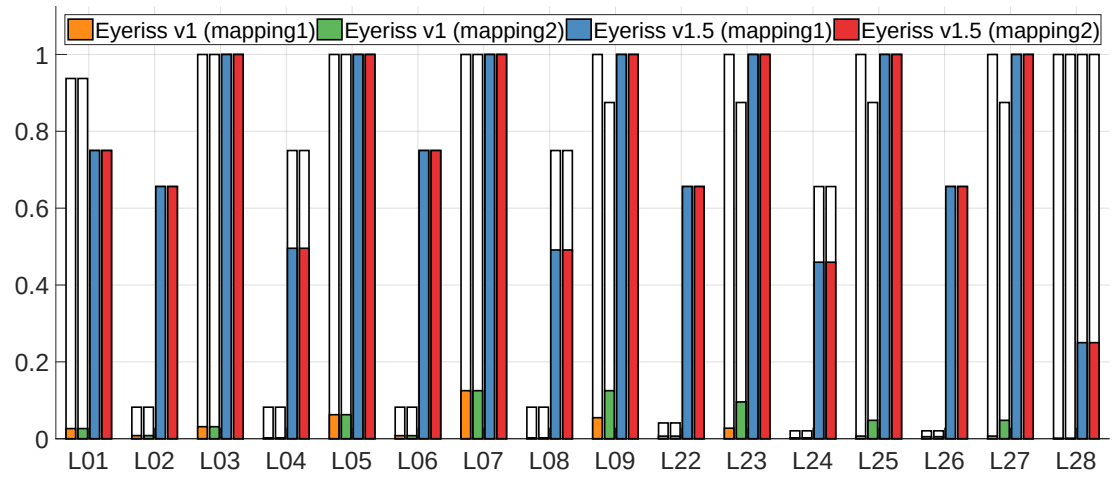


(c) 16384 PEs

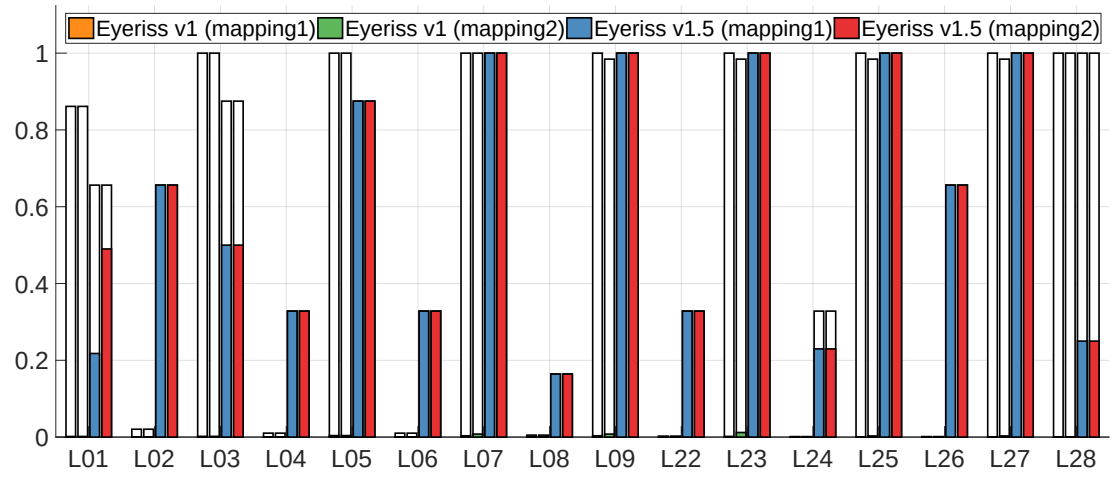
Figure 5-17: Performance of GoogLeNet at PE array size of (a) 256, (b) 1024 and (c) 16384. Performance is normalized to the peak performance.



(a) 256 PEs



(b) 1024 PEs



(c) 16384 PEs

Figure 5-18: Performance of MobileNet at PE array size of (a) 256, (b) 1024 and (c) 16384. Performance is normalized to the peak performance.

For each combination of the architecture and DNN layer, we generate the optimal mappings for the following two different objectives:

- *Mapping 1* is optimized to get the highest number of active PEs regardless the actual utilization of them, i.e., it is optimized for the overall bar height (white + color).
- *Mapping 2* is optimized to get the best overall utilization of the PE array that further accounts for the impact of the finite bandwidth on performance, i.e., it is optimized for the height of the colored-only bar.

First, we compare only the number of active PEs, i.e., total bar height, of the two architectures. In most cases, Eyeriss v1.5 shows a better performance than Eyeriss v1 except for a few cases at the PE array size of 16384. In these cases, the performance degradation in Eyeriss v1.5 is because that the number of clusters becomes too large while the cluster size is kept small. A small cluster size ensures that the implementation cost of the all-to-all network is limited. As mentioned in Section 5.4, the hierarchical mesh network imposes mapping constraints due to its two-level structure. It requires a high amount of reuse in one data type to fully map the large number of clusters, while the other data type can only exploit reuse up to the amount of the cluster size. Eventually, this can lead to spatial mapping fragmentation, which reduces the number of active PEs. Eyeriss v1, on the other hand, may adapt better at large PE array sizes in terms of number of active PEs since multiple layer dimensions can be mapped onto the same physical dimension of its PE array even with less number of data dimensions to choose from for spatial mapping. This phenomenon is more significant in GoogLeNet than in AlexNet, since the layers in AlexNet are usually much larger than that in GoogLeNet and have plenty of use for all data types, which results in less mapping fragmentation. In general, mapping fragmentation is also less severe in smaller PE array sizes, and the more flexible spatial mapping of the RS+ dataflow gives Eyeriss v1.5 an edge over Eyeriss v1. Specifically, the RS+ dataflow handles the mapping in the DW CONV layers of MobileNet much better than the RS dataflow as shown in the even layers of MobileNet (except for L28, which is a FC layer).

Next, we compare the performance in terms of the overall utilization of the PE array, which is the colored-only portion of the bars. This comparison shows a drastic performance

difference between the two architectures. For Eyeriss v1.5, most of the time it can keep the overall utilization of the PE array up to the level of the number of active PEs. However, for Eyeriss v1, there is usually a big gap between the number of active PEs and the overall utilization of the PE array. The performance difference grows even higher when the PE array size scales. In certain cases, however, Eyeriss v1.5 still loses performance due to the finite bandwidth. For example, in the FC layers, e.g., L6 to L8 in AlexNet, L58 in GoogLeNet and L28 in MobileNet, the overall utilization of the PE array is only one fourth of the number of active PEs. The performance bottleneck comes from the insufficient bandwidth for delivering weights. In our setup, each weight router cluster has 4 routers and connects to 4 GLB banks and 16 PEs, which means that 4 PEs in a cluster share the same GLB bank for weight delivery. In the case when batch size is 1, there is no weight reuse, and therefore the performance is reduced by 4 times.

Finally, we compare the impact of having the mapping being optimized for different objectives. For Eyeriss v1, mapping 1 usually results in a higher number of active PEs than mapping 2; however, mapping 2 still shows a higher overall utilization of the PE array than mapping 1. This shows that optimizing for the maximum number of active PEs does not necessarily yield the best performance after considering the finite bandwidth, especially when the deliverable bandwidth is low. This is mainly because the mapping often relies on certain layer dimensions that can provide a large tile to fully utilize the high parallelism. However, this also results in a higher bandwidth requirement for a specific data type. Instead, optimizing the mapping according to the actual overall utilization of the PE array, mapping 2 takes the bandwidth constraints into account and avoids placing too much pressure on the bandwidth of certain data types. For example, in AlexNet L01 with a PE array size of 256, mapping 1 relies on a large number of output channels to fill the parallelism, which results in a high bandwidth requirement for weights, while mapping 2 has a more balanced tile size between the input and output channels, and therefore distributes the bandwidth requirement between weights and input activations. This phenomenon, however, is less prominent in Eyeriss v1.5. Since Eyeriss v1.5 can provide a more scalable data bandwidth, optimizing for the number of active PEs is usually enough to guarantee a high overall utilization of the PE array.

Table 5.3 quantifies the performance speedup of Eyeriss v1.5 over Eyeriss v1 in terms of the overall utilization of the PE array. For each combination of DNN and PE array size, it shows the range of speedup along with the averages across the layers in the same DNN. At the end of each column, it also shows the average speedup across all layers in the three DNNs at the same PE array size. The FC layers achieve the highest speed up, as the increase in bandwidth for delivering weights has a significant impact on performance. Another highlight is the performance of the DW CONV layers in MobileNet, which receives a speedup ranging from 25.4× (256 PEs) to 997.5× (16384 PEs).

		256 PEs	**1024 PEs**	**16384 PEs**
AlexNet	range	4.3×–64.0×	16.8×–256.0×	80.0×–4096.0×
	average	33.1×	132.4×	2082.6
	weighted average	17.9×	71.5×	1086.7×
GoogLeNet	range	1.8×–64.0×	9.1×–256.0×	13.1×–4096.0×
	average	17.3×	65.7×	757.0×
	weighted average	10.4×	37.8×	448.8×
MobileNet	range	3.0×–64.0×	8.0×–256.0×	22.4×–4096.0×
	average	26.1×	101.0×	1083.2×
	weighted average	15.7×	57.9×	873.0×
Overall	average	21.3×	81.9×	967.0×
	weighted average	13.3×	50.3×	693.3×

Table 5.3: Performance Speedup of Eyeriss v1.5 over Eyeriss v1. The average speedup simply takes the mean of speedups from all layers in a DNN; the weighted average is calculated by weighting the speedup of each layer with the proportion of MACs of that layer in the entire DNN.

5.6 Conclusions

DNNs are rapidly evolving due to the significant amount of research in the field; however, the current direction of DNN development also brings new challenges to the DNN accelerator design due to the higher variation in data reuse among different DNNs. In this chapter,

we propose Eyexam, a performance analysis framework that provides a seven step process to systematically identify the sources of performance loss in a DNN accelerator and can be used to develop a set of roofline models to assess the impact of bandwidth constraints. Eyexam gives insights into the performance bottleneck in existing DNN accelerators and inspires the design of Eyeriss v1.5. Eyeriss v1.5 achieves high flexibility through a new dataflow, called row-stationary plus (RS+), that can maintain high utilization of PEs for both large and compact DNNs that have a wide range of data reuse. To support RS+, it also features a flexible NoC, called hierarchical mesh, that can provide high bandwidth when data reuse is low while still being able to exploit high data reuse when available. Overall, Eyeriss v1.5 achieves more than a 10$\times$ throughput speedup over Eyeriss v1 at 256 PEs, and this performance advantage increases when the architecture scales to a higher number of PEs.

Chapter 6

Eyeriss v2

In this chapter, we introduce Eyeriss v2, a DNN accelerator architecture designed for the processing of both large and compact DNNs, which have a wide range of data reuse. Eyeriss v2 supports the RS+ dataflow and is built on the hierarchical mesh network introduced in Chapter 5 and demonstrated on Eyeriss v1.5. In addition, it also leverages data sparsity and SIMD to further improve performance and energy efficiency. We will first give an overview of the architecture in Section 6.1, and then dive into the architecture implementation details, including the hierarchical mesh network (Section 6.2), the PE architecture that supports the RS+ dataflow and can further leverage data sparsity (Section 6.3), and SIMD processing (Section 6.4). Finally, we will present the implementation results in Section 6.5.

6.1 Architecture Overview

Eyeriss v2 is built on the hierarchical mesh network, which is illustrated in Fig. 5-15. Each PE cluster in Eyeriss v2 has 12 PEs arranged in a 3×4 array, and there are 16 PE clusters arranged in a 8×2 cluster array. In total, there are 192 PEs. Each PE cluster connects to three router clusters for the delivery of three data types (i.e., input activations, weights and psums). The three router clusters are then connected to a single global buffer (GBuff) cluster. Each router cluster is also connected to its neighbor router clusters of the same data type. The router cluster for input activations, weights and psums have 3, 3, and 4 routers, respectively. Each GBuff cluster is a multi-bank memory, in which the memory banks can be allocated at

compile time for the storage of the three data types and provide 3, 3, and 4 R/W ports for accessing input activations, weights and psums, respectively. The architecture of each GBuff cluster is similar to the global buffer implemented in Eyeriss v1 (Section 4.4.1). The storage capacity of each GBuff cluster is 11.25 kB; therefore, the total GBuff storage capacity in Eyeriss v2 is 180 kB.

Eyeriss v2 is designed to process 8b fixed-point weights and input activations. However, the psums are kept at 20b. Note that the 8b input activations can be set to be either signed or unsigned through one configuration bit for an entire DNN layer. For example, the layers after ReLU will have unsigned input activation, and therefore can take advantage of the unsigned representation to improve precision.

6.2 Implementation of the Hierarchical Mesh Networks

In this section, we will describe the implementation of the hierarchical mesh networks for the three data types. Specifically, we will discuss how to reduce the implementation complexity of the network and how it affects the mapping space of the RS+ dataflow.

The 2D mesh connections of the router clusters and the all-to-all network between the router cluster and PE cluster are required to guarantee that data can be delivered from any port in any GBuff cluster to any PE. It is critical for the support of all four data delivery patterns as shown in the 1D example in Fig. 5-14. However, we find it possible to relax this condition when growing the network in 2D by treating the 2D network as many 1D ones for specific data types, which reduces the implementation complexity.

For example, for the six data dimensions that can be mapped spatially in the RS+ dataflow, i.e., $G1$, $N1$, $M1$, $E1$, $C1$ and $R1$ in Fig. 5-8, some of them can be restricted to be mapped onto just one physical dimension of the PE array. For instance, if $F1$ and $N1$ are not allowed to be mapped onto the vertical dimension of the PE array, there will be no vertical spatial reuse of weights. Therefore, weights from any one of the three ports of the GBuff cluster only need to reach PEs on the corresponding row in a PE cluster. The optimized weight hierarchical mesh network is shown in Fig. 6-1.

Similarly, if data dimensions $R1$ and $C1$ are not allowed to be mapped onto the horizontal

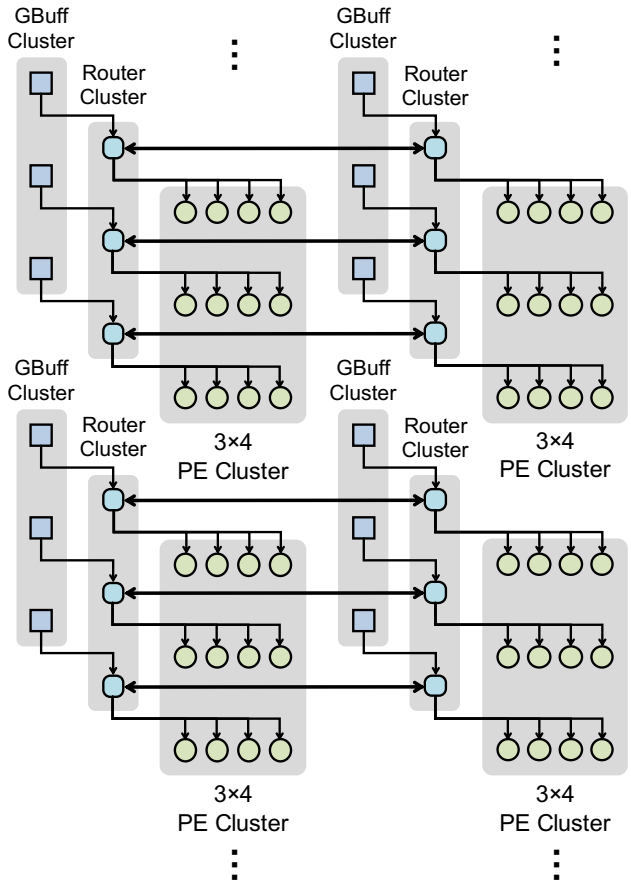


Figure 6-1: Weight hierarchical mesh network in Eyeriss v2. All vertical connections are removed after optimization.

dimension of the PE array, there will be no horizontal accumulation of psums. Therefore, psums from any one of the four ports of the GBuff cluster only need to read PEs on the corresponding column in a PE cluster. In addition, in order to take advantage of local psum accumulation, direct vertical PE-to-PE links are added in between PEs within a cluster and across clusters. Therefore, each PE cluster can select input psums either from the router cluster or from its neighbor PE cluster. The router cluster can receive input psums from any GBuff cluster on the same column. The output psums generated from a PE cluster can also select its destination to either its neighbor PE cluster or back to the router cluster, which then send the psums back to the corresponding GBuff clusters. The optimized psum hierarchical mesh network is shown in Fig. 6-2. The routing patterns are all deterministic and are configured at compile time.

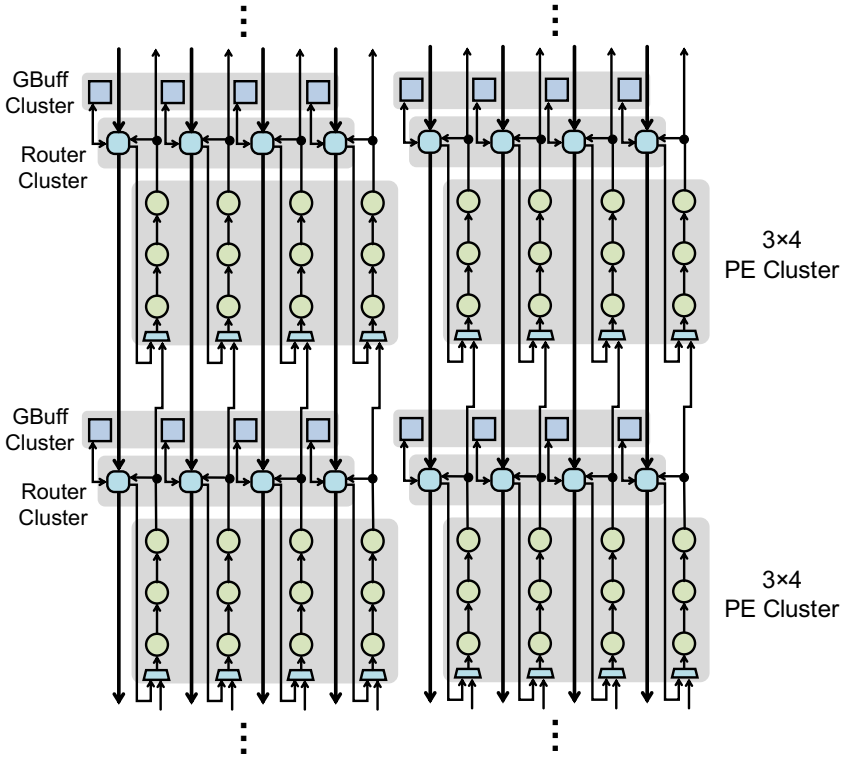


Figure 6-2: Psum hierarchical mesh network in Eyeriss v2. All horizontal connections are removed after optimization.

For the input activations, we found that it is important to keep the 2D mesh and the all-to-all connections in order to maintain the performance of the RS+ mappings. Therefore, the hierarchical mesh network described in Section 5.4 is implemented. Each router cluster is connected to all of its four neighbor router clusters, and the 3 routers in the router cluster are connection to all 12 PEs in the PE cluster.

6.3 Exploiting Data Sparsity

In Eyeriss v1, data sparsity of input activations, i.e., zeros, is exploited to improve energy efficiency. In Eyeriss v2, we want to exploit sparsity in both weights and input activations to improve not only energy efficiency but also performance. In addition, the processing should directly take advantage of the compressed data representation. In this section, we will introduce a new PE architecture that can support the mappings of the RS+ dataflow

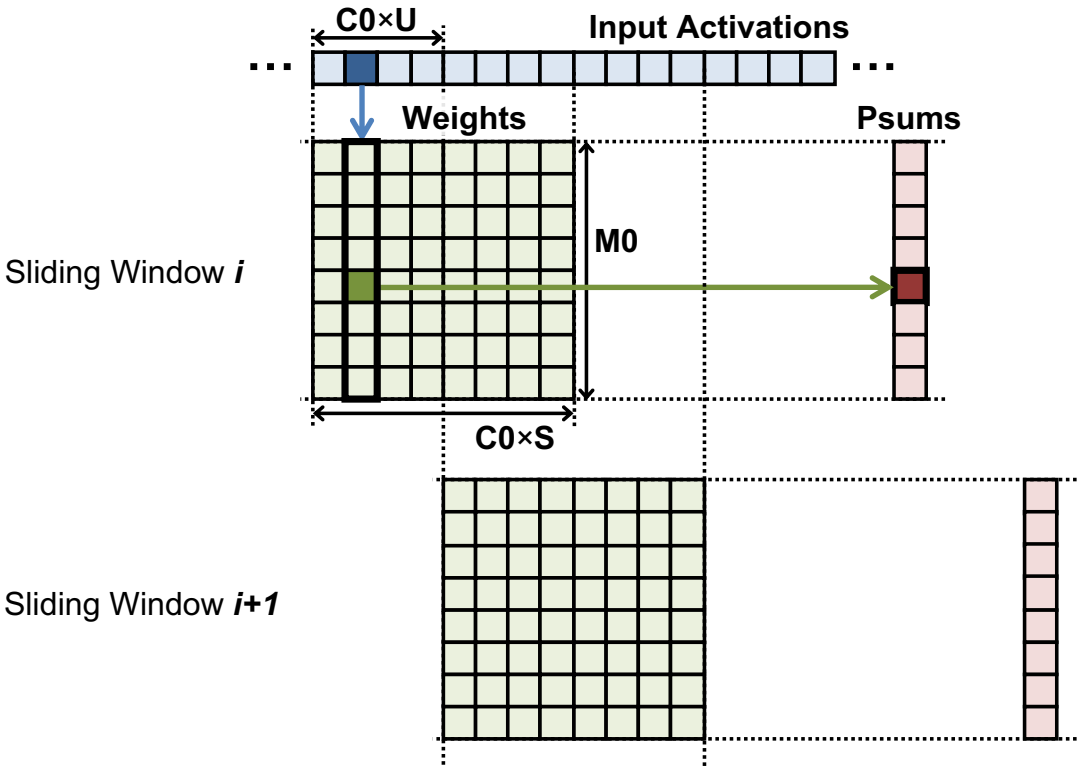


Figure 6-3: Processing mechanism in the PE.

while further achieving processing in the compressed domain to exploit sparsity for higher performance.

Fig. 6-3 illustrates how the PE processes uncompressed weights and input activations given mappings from the RS (and RS+) dataflow. For each input activation, the PE runs through $M0$ MAC operations sequentially in consecutive cycles with the corresponding column of $M0$ weights in the weight matrix, and accumulates to $M0$ psums. By going through a window of $C0 \times S$ input activations in the stream, the processing goes through all $M0 \times C0 \times S$ weights in the matrix and accumulates to the same $M0$ psums. It then slides to the next window in the input activation stream by replacing $C0 \times U$ input activations at the front of the window with new ones, where U is the stride, and repeats the processing with the same $M0 \times C0 \times S$ weights but accumulates to another set of $M0$ psums. Note that the access pattern of weights goes through the entire weight matrix once sequentially in a column-major fashion for each window of input activations.

To speedup the processing when the input activations and/or weights are sparse, the

goal is to be able to read only the non-zero data in the input activation stream and the weight matrix for processing. The challenge, however, is to correctly and efficiently address and access data from all three data types. For example, when jumping between non-zero input activations in a window, the access pattern of weights does not go through the weight matrix sequentially anymore. Instead, additional logic is required to fetch the corresponding column of weights for the non-zero input activation, which is not deterministic. Similarly, when jumping between non-zero weights in a weight column, it also has to calculate the address of the corresponding psum instead of just incrementing the address by one.

In order to achieve the processing of sparse data as described above, we take advantage of the compressed sparse column (CSC) compression format similar to what is described in [17, 24]. For each non-zero value in the data, the CSC format records an additional *count* value that indicates the number of leading zeros from the previous non-zero value in the uncompressed data stream. The count value can then be used to calculate the address change between the non-zero data.

For input activations, the data stream is divided into non-overlapping $C0 \times U$ segments, and each segment is CSC encoded separately. Doing so enables the sliding window processing, which replace a $C0 \times U$ segment of the data stream with a new segment when the window slides. Since the data length of each segment will be different after CSC coding, additional information is needed to address each encoded segment. Therefore, for each encoded segment, an *address* value is also recorded in the CSC format that indicates the start address of the encoded segment in the entire encoded stream. The filter weights are also encoded with CSC compression by dividing each column of $M0$ weights as a segment and encoding each segment separately. This helps to access each column of non-zero weights quickly.

Fig. 6-4 shows an example of CSC compressed weights. The characters in the weight matrix indicate the locations of the non-zero data. To read the non-zero weights from a specific column, e.g., column 1 (assuming indexing starts from 0), the PE first reads address[1] and address[2] from the address vector in the CSC compressed weights, which gives the inclusive lower bound and non-inclusive upper bound of the addresses, i.e., 2 and 5, respectively, for reading the data and count vector. It then goes through the three

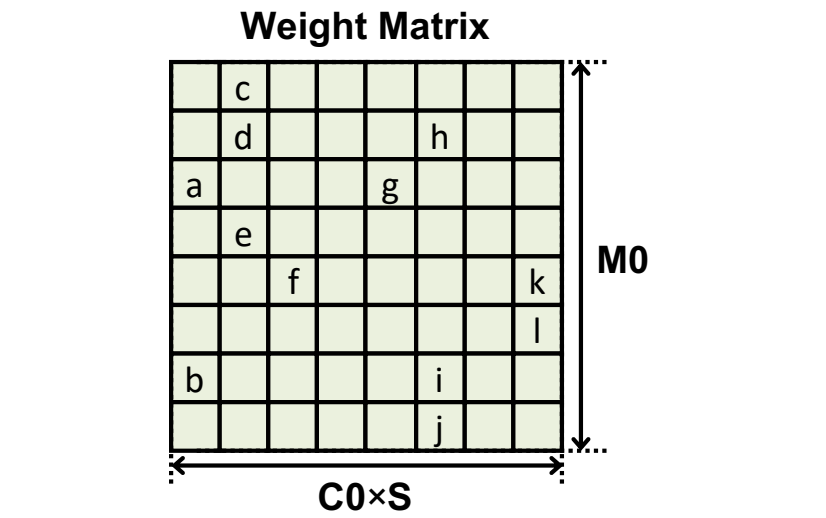


Figure 6-4: Example of compressing sparse weights with compressed sparse column (CSC) coding.

non-zero weights in the column, i.e., *c*, *d* and *e*, to perform the computation. At the same time, the corresponding addresses of the psums to update can be calculated by accumulating the counts from the count vector.

In summary, both the weights and input activations can be processed directly in the CSC format. The processing can skip the zeros entirely without spending extra cycles, thus improving the processing throughput as well as energy efficiency.

Fig. 6-5 shows the block diagram of the *sparse PE* that can perform the processing of CSC encoded input activations and weights directly as described above. The PE has 7 pipeline stages and 5 SPads. The first 2 pipeline stages are responsible for fetching non-zero input activations from the SPads. The input activation (iact) address SPad stores the address vector of the CSC compressed input activations, which is used to address the reads from the iact data SPad that holds the non-zero data vector as well as the count vector. After a non-zero input activation is fetched, the next 3 pipeline stages read the corresponding

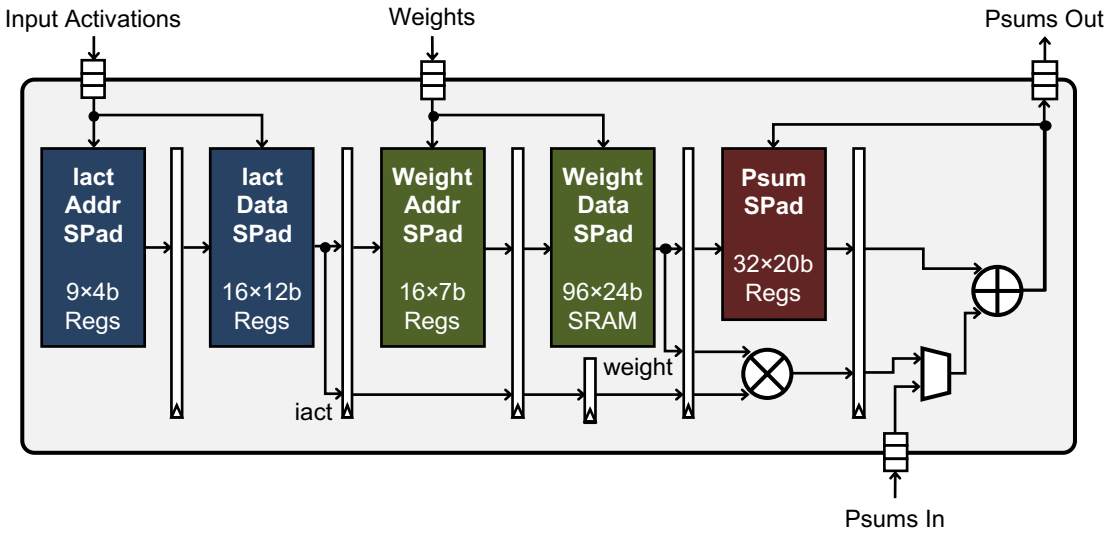


Figure 6-5: Eyeriss v2 PE Architecture

weights. Similarly, there is a weight address SPad to address the reads from the weight data SPad for the correct column of weights. The final 2 stages in the pipeline perform the MAC computation on the fetched non-zero input activation and weight, and then send the updated psum either back to the psum SPad or out of the PE.

In the CSC format, the count vector is a major overhead in addition to the non-zero data. If the bitwidth of the count is low, it may affect the compression efficiency when sparsity is high since the number of consecutive zeros can exceed the maximum count. If the count bitwidth is high, however, the overhead of the count vector becomes more significant. From our experiments, setting each count at 4b yields the best compression rate for the 8b input activations and weights. Therefore, each count-data pair is 12b. This is similar to setting the run-length in the RLC coding used in Eyeriss v1 as discussed in Section 4.3. In the previous example, however, a longer run-length, i.e., count, of 5b is used since the data is 16b instead of 8b.

In the Eyeriss v2 PE, the sizes of the input activation address and data SPads are 9×4b and 16×12b, respectively. The sizes of the weight address and data SPads are 16×7b and 96×24b, respectively. The size of the psum SPad is 32×20b. This allows the mapping with a maximum $M0$ of 32 and a maximum $C0 \times S$ of 15. Note that the weight data SPad can

only hold a maximum of 192 count-data pairs, which is less than 32×15=480. This design takes advantage of the fact that the sparse pattern of the weights is known at compile time; therefore, it is possible to guarantee that the compressed weights will fit in a smaller SPad.

Since the degree of sparsity varies across different DNNs and input data, the PE is also designed to adapt to the scenarios when sparsity is low. In such cases, the PE can directly take in uncompressed input activations and weights instead of the CSC compressed versions to reduce the overhead in data traffic. Also, both the iact and weight address SPads are clock-gated to save energy consumption. Additional logic is used to address the iact and weight data SPad.

6.4 Exploiting SIMD Processing

Profiling results of the PE implementation shows that the area and energy consumption of the MAC unit is insignificant compared to other components in a PE. In Eyeriss v1, for example, the MAC unit takes less than 5% of the PE area, and only consumes 2%–9% of the PE power. This motivates the exploration of SIMD processing in a PE in order to achieve higher performance.

SIMD is applied to the PE architecture shown in Fig. 6-5 by fetching two weights instead of one for computing two MAC operations per cycle with the same input activation. Therefore, the SIMD width is 2. SIMD processing not only improves the throughput but also further reduces the number of input activation reads from the SPad. In terms of architectural changes, SIMD requires the word width of the weight data SPad to be two-words wide, which is why the size of the weight data SPad is 96×24b instead of 192×12b. The psum SPad also has two read and two write ports for updating two psums per cycle. In the case where only an odd number of non-zero weights exist in the column of $M0$ weights, the second 12b of the last 24b word in a column of non-zero weights is filled with zeros. When the PE logic encounters the all-zero count-data pair, it clock-gates the second MAC datapaths as well as the read and write of the second ports in the psum SPad to save unnecessary switching power consumption.

6.5 Implementation Results

The Eyeriss v2 architecture was implemented in a 65nm CMOS process. The design was placed-and-routed and the results reported in this section are from post-layout cycle-accurate gate-level simulations with (1) technology library from the worst PVT corner, and (2) switching activities profiled from running the actual weights of the DNNs and data from the ImageNet dataset [56].

The implemented design consists of the PEs (Section 6.3) and the hierarchical mesh network for all data types (Section 6.2). The global buffer is not included in this result, and we leave its implementation for future endeavors. Instead, we focus on the analysis of NoC and PEs since it's the main innovation of Eyeriss v2 from Eyeriss v1.

The overall gate counts of the implemented Eyeriss v2, excluding SRAMs, is approximately 2655k NAND-2 gates. The area breakdown shows that the 192 PEs dominates the area cost, while the area of the hierarchical mesh networks of all data types combined only account for 3.5% of the area. This result proves that it is possible to build in high flexibility at a low cost. Fig. 6-6 shows the area breakdown for the Eyeriss v2 PE. All of the SPads combined account for almost 75% of the PE area, while the two MAC units only account for 5%. Among the SPads, the psum SPad is the largest one even though it does not have the largest storage capacity. This is due to the need to support two read and two write ports to support SIMD processing.

Table 6.1, 6.2 and 6.3 show the performance and energy efficiency of running AlexNet [36], sparse-AlexNet [71] and MobileNet [30] on Eyeriss v2, respectively. The results can be summarized as follows:

- While Eyeriss v2 can already exploit any sparsity in the original AlexNet, the performance and energy efficiency are both much further improved when running the sparse-AlexNet, which has around 90% zeros in the weights.

- The energy efficiency varies widely across the layers of the three DNNs. While sparsity contributes a lot, the amount of data reuse still plays a key role. For example, in sparse-AlexNet, even though the FC layers have at least on par, if not higher, amount of sparsity than the CONV layers, the lack of reuse in the end still limits the

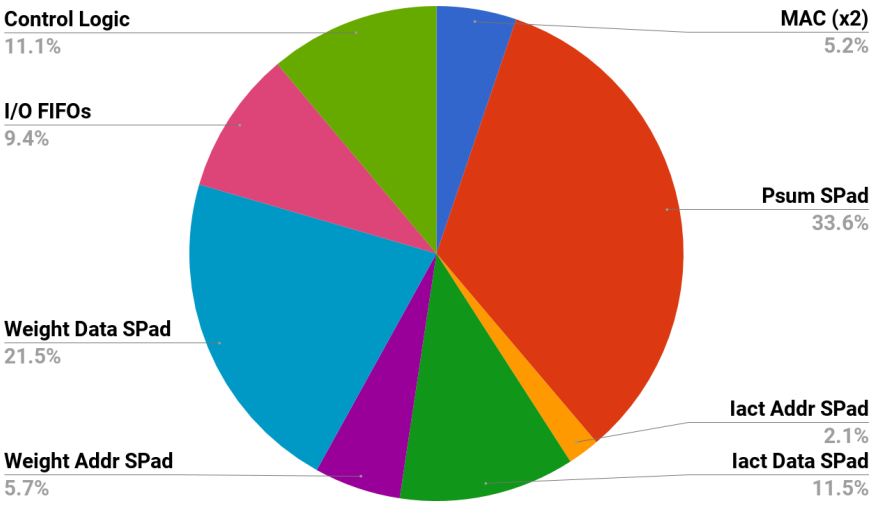


Figure 6-6: Area breakdown of the Eyeriss v2 PE.

energy efficiency of FC layers. Overall, CONV3 in the sparse-AlexNet achieves the highest energy efficiency due to its high data sparsity and high reuse.

- The depth-wise (DW) CONV layers in MobileNet in general have the worst energy efficiency for two reasons: (1) they have little data reuse due to the lack of input and output channels; (2) the lack of output channels also renders the SIMD feature useless. The overhead of supporting SIMD, such as the 2-word weight data SPad and the multi-R/W-port psum spad, also reduces the energy efficiency.

- The performance in terms of GOPS and the energy efficiency in term of GOPS/W of MobileNet are the lowest among the three DNNs. However, from an application point of view, the more relevant metrics will be evaluating performance in in terms of frames per second (fps) and energy efficiency in terms of number of inferences per joule (Inf/J). In this case, MobileNet has the highest performance and energy efficiency. This is due to the fact that it takes much lower number of operations to complete a pass of inference in MobileNet than in AlexNet.

Fig. 6-7 shows the power breakdown of Eyeriss v2 running a variety of DNN layers. We pick a representative set of layers to show how the different characteristics of the DNN layers impact the hardware. The results are summarized as follows:

Layer	# of MACs	Latency (ms)	Power (mW)	Performance (GOPS)	Efficiency (GOPS/W)
CONV1	105.4M	1.53	915.3	137.5	150.3
CONV2	223.9M	3.15	605.1	142.3	235.1
CONV3	149.5M	1.57	423.6	190.1	448.8
CONV4	112.1M	1.03	437.4	217.0	496.1
CONV5	74.8M	0.66	435.3	227.2	522.0
FC6	37.7M	1.06	293.4	71.4	243.3
FC7	16.8M	0.58	255.4	57.4	224.9
FC8	4.1M	0.18	225.1	45.6	202.3
Overall	724.4M	9.77	533.7	148.3 (102.4 fps)	277.9 (191.8 Inf/J)

Table 6.1: Performance and Energy Efficiency of running AlexNet [36] on Eyeriss v2. The results are from post-layout gate-level cycle-accurate simulations at the worst PVT corner with a batch size of 1 and a clock rate of 200 MHz.

Layer	# of MACs	Latency (ms)	Power (mW)	Performance (GOPS)	Efficiency (GOPS/W)
CONV1	105.4M	0.74	482.1	283.7	588.4
CONV2	223.9M	0.82	470.0	549.4	1169.0
CONV3	149.5M	0.53	362.9	562.0	1548.2
CONV4	112.1M	0.43	415.2	519.0	1249.8
CONV5	74.8M	0.38	414.2	395.3	954.3
FC6	37.7M	0.39	215.1	195.2	907.4
FC7	16.8M	0.19	200.2	172.4	861.3
FC8	4.1M	0.09	186.0	93.2	501.3
Overall	724.4M	3.57	394.7	405.8 (280.1 fps)	1028.1 (709.7 Inf/J)

Table 6.2: Performance and Energy Efficiency of running Sparse-AlexNet [71] on Eyeriss v2. The results are from post-layout gate-level cycle-accurate simulations at the worst PVT corner with a batch size of 1 and a clock rate of 200 MHz.

Layer	# of MACs	Latency (μs)	Power (mW)	Performance (GOPS)	Efficiency (GOPS/W)
CONV1	1.77M	29.0	512.4	122.1	238.3
CONV2 DW	0.59M	22.0	455.1	53.5	117.6
CONV2 PW	2.10M	42.9	455.1	97.8	214.9
CONV3 DW	0.29M	11.9	465.1	49.7	106.8
CONV3 PW	2.10M	26.4	756.4	159.0	210.2
CONV4 DW	0.59M	23.6	454.5	50.0	110.1
CONV4 PW	4.19M	54.2	800.2	154.8	193.5
CONV5 DW	0.15M	6.8	432.0	43.6	100.9
CONV5 PW	2.10M	29.1	829.6	143.9	173.5
CONV6 DW	0.29M	13.4	425.6	44.1	103.6
CONV6 PW	4.19M	52.8	744.5	159.0	213.6
CONV7 DW	0.07M	4.2	354.9	34.9	98.4
CONV7 PW	2.10M	30.8	779.3	136.2	174.8
CONV8 DW	0.15M	8.3	373.8	35.4	94.6
CONV8 PW	4.19M	53.2	720.1	157.8	219.2
CONV9 DW	0.15M	8.3	362.5	35.4	97.5
CONV9 PW	4.19M	52.1	721.5	161.2	223.3
CONV10 DW	0.15M	8.3	347.2	35.4	101.8
CONV10 PW	4.19M	53.3	704.9	157.3	223.1
CONV11 DW	0.15M	8.3	364.0	35.4	97.1
CONV11 PW	4.19M	52.9	740.2	158.6	214.3
CONV12 DW	0.15M	8.3	358.7	35.4	98.6
CONV12 PW	4.19M	53.5	698.8	156.9	224.6
CONV13 DW	0.04M	2.9	305.4	25.4	83.2
CONV13 PW	2.10M	34.0	635.0	123.3	194.2
CONV14 DW	0.07M	5.6	290.7	26.2	90.2
CONV14 PW	4.19M	61.5	606.4	136.5	225.0
FC15	0.51M	20.5	361.6	50.0	138.4
Overall	49.2M	778.1	636.6	126.4 (1285.2 fps)	198.5 (2020.8 Inf/J)

Table 6.3: Performance and Energy Efficiency of running MobileNet [30] on Eyeriss v2. The results are from post-layout gate-level cycle-accurate simulations at the worst PVT corner with a batch size of 1 and a clock rate of 200 MHz.

- CONV1 of AlexNet (Fig. 6-7a) is picked to examine the case when there is no sparsity in the data. Compared to other layers, the high utilization of the PEs makes the proportion of the clock network power consumption low. It also has the highest proportion of MAC power consumption.

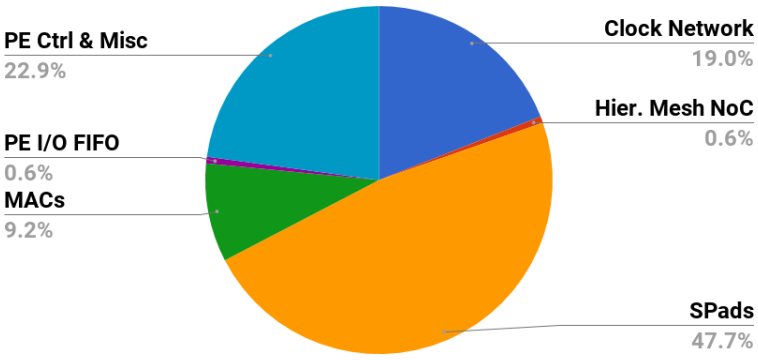
- CONV3 of sparse-AlexNet (Fig. 6-7b) is picked since it has the highest energy efficiency among all layers. The proportion of the clock network power consumption is higher than that of CONV1 of AlexNet. This is mainly due to the workload imbalance induced by sparsity, which lowers the utilization of the active PEs. However, judging from the large proportion of the SPad and MAC power consumption compared to other components such as PE control logic, the PE is still kept fairly busy and data reuse is effectively exploited by the SPads, which contributes to the high overall energy efficiency.

- CONV13 DW CONV layer (Fig. 6-7c) of MobileNet is picked due to its lowest energy efficiency among all layers. As expected, most of the energy is wasted on the clock network. Inside the PE, the lack of reuse and not being able to utilize SIMD have also hurt the energy efficiency, which is evident by the fact that most of the energy is spent in the control logic instead of the SPads or MACs.

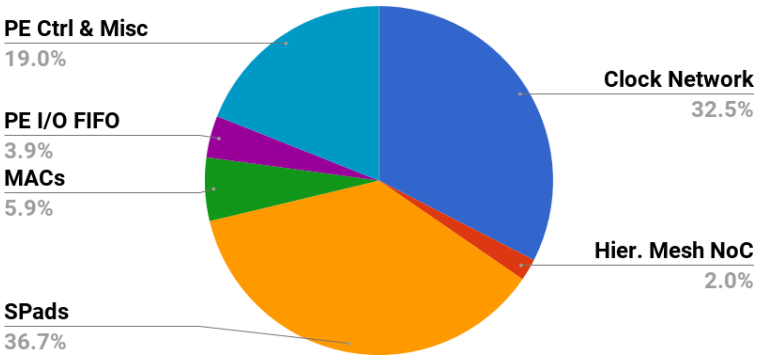
- FC8 of sparse-AlexNet (Fig. 6-7d) is picked to examine the case when sparsity is high but data reuse is low. This combination makes the architecture more bandwidth-limited, and therefore the utilization of active PEs becomes low. That is why this layer has the highest proportion of power consumed by the clock network. The lack of reuse also makes the proportion of the SPad power consumption low and the NoC power consumption high. However, thanks to sparsity, the overall energy efficiency of this layer is still better than CONV1 of AlexNet.

6.6 Comparison of Different Eyeriss Versions

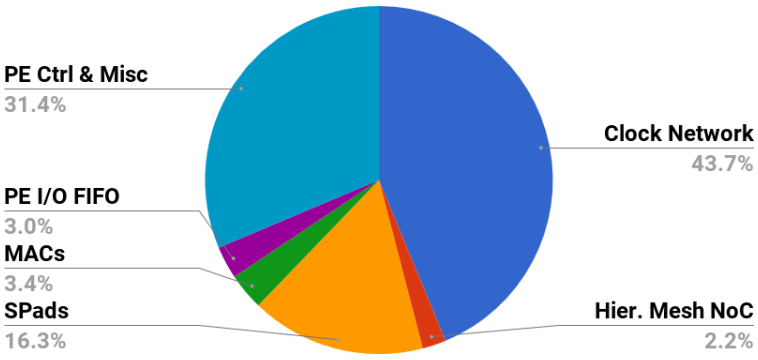
In this section, we compare the Eyeriss v2 architecture with Eyeriss v1 and v1.5 (Chapter 5) in terms of performance and energy efficiency. Table 6.4 summaries the key specs and



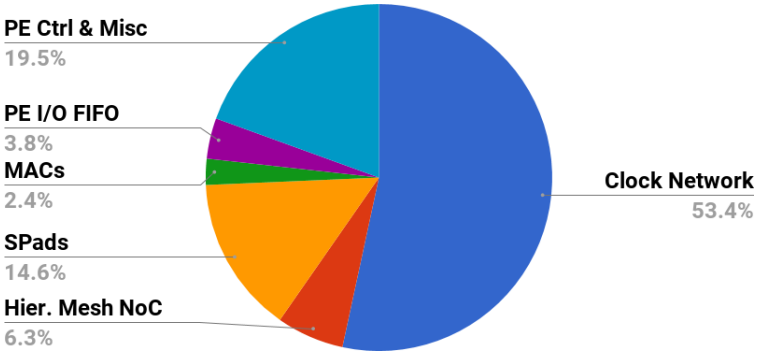
(a) CONV1 of AlexNet



(b) CONV3 of sparse-AlexNet



(c) CONV13 DW of MobileNet



(d) FC8 of sparse-AlexNet

Figure 6-7: Power breakdown of Eyeriss v2 running a variety of DNN layers.

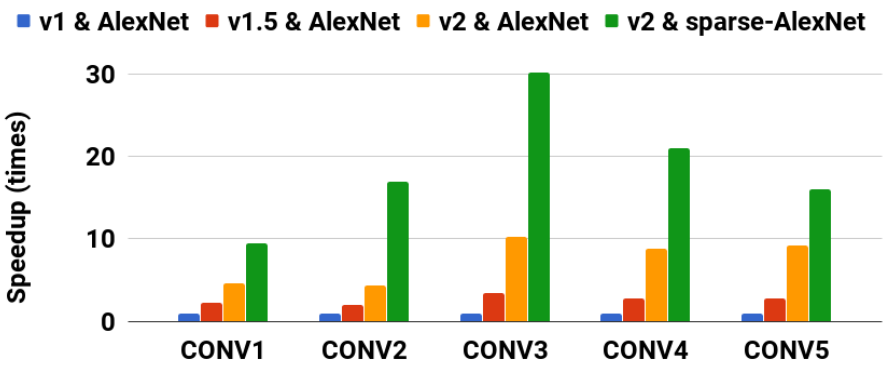
	Eyeriss v1	Eyeriss v1.5	Eyeriss v2
Data Bitwidth	weight/activation: 8b, psum: 20b		
# of PEs	192		
# of MACs	192	192	384
SPad size/PE	0.3 kB		
Total GBuff Size	180 kB		
Clock Rate	200 MHz		
Dataflow	RS	RS+	RS+
NoC	Multicast	Hier. Mesh	Hier. Mesh
PE Architecture	Dense	Dense	Sparse
SIMD Support	No	No	Yes

Table 6.4: Key specs of the three Eyeriss variants. For the PE architecture, dense means it can only clock-gate the cycles with zero data but not skip it, while the sparse means it can further skip the processing cycles with zero data (Section 6.3).

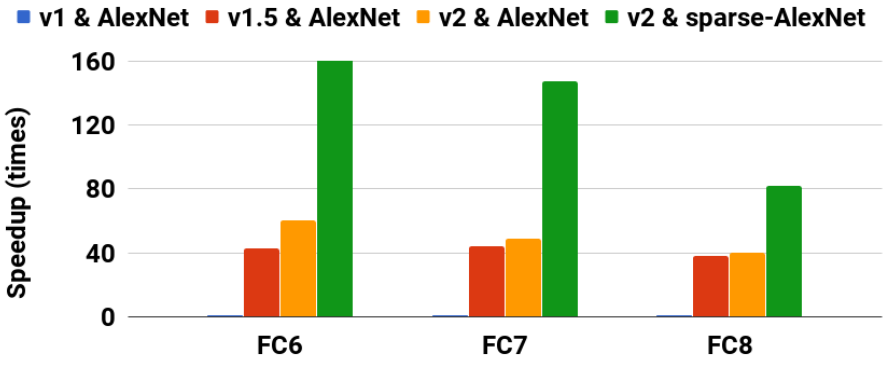
differences of the three Eyeriss variants. Note that we adapt the data bitwidth and storage capacity of Eyeriss v1 in order to make a fair comparison and show the improvements from each architectural change. The simulations use the same setup as in Section 6.5.

Fig. 6-8 shows the performance speedup between different versions of Eyeriss on AlexNet. Note that sparse-AlexNet is also included in the comparison (see green bar). The result shows that Eyeriss v1.5 significantly speeds up FC layers. This is because the performance of FC layers is bandwidth-limited in Eyeriss v1. Eyeriss v2, on the contrary, significantly speeds up the CONV layers over Eyeriss v1.5, while the performance of the FC layers only shows a marginal improvement. This is because the FC layers are still bandwidth-limited even with the hierarchical mesh network; therefore, speeding up the processing with sparsity does not improve the throughput of FC layers as significantly as in CONV layers. The full potential of Eyeriss v2, however, is fully revealed when coupled with sparse-AlexNet. The bandwidth requirement of weights is lower in sparse-AlexNet since it is very sparse, and the CSC compression can effectively reduce the data traffic. As a result, exploiting sparsity becomes more effective. Overall, Eyeriss v2 achieves 42.5× performance speedup with sparse-AlexNet over Eyeriss v1 with AlexNet.

Fig. 6-9 shows the improvement on energy efficiency. It largely correlates to the speedup in Fig. 6-8 since the higher overall utilization of the PEs reduces the proportion of the static power consumption, e.g., clock network. Overall, Eyeriss v2 with sparse-AlexNet is 11.3×



(a) CONV Layers of AlexNet



(b) FC Layers of AlexNet

Figure 6-8: Performance speedup between different Eyeriss variants on AlexNet. Eyeriss v1 is used as the baseline for all layers. The experiment uses a batch size of 1.

more energy efficient than Eyeriss v1 with AlexNet.

Fig. 6-10 shows the performance speedup between different versions of Eyeriss on selected layers of MobileNet. As discussed in Chapter 5, the lack of data reuse in MobileNet results in low performance on Eyeriss v1 due to the low-bandwidth NoC, which is why Eyeriss v1.5 can achieve a significant speedup over v1. However, the speedup of Eyeriss v2 over v1.5 is a mixed bag. While layers such as CONV1 and the point-wise (PW) layers can still take advantage of the sparsity in input activations to improve the performance, the performance of the DW CONV layers actually goes worse. This is because the CSC compression does not work when the number of input and output channels are both 1. Therefore, the sparse PE in Eyeriss v2 does not bring any advantage over the dense PE in Eyeriss v1.5. Furthermore, the deeper pipeline of the sparse PE actually makes the

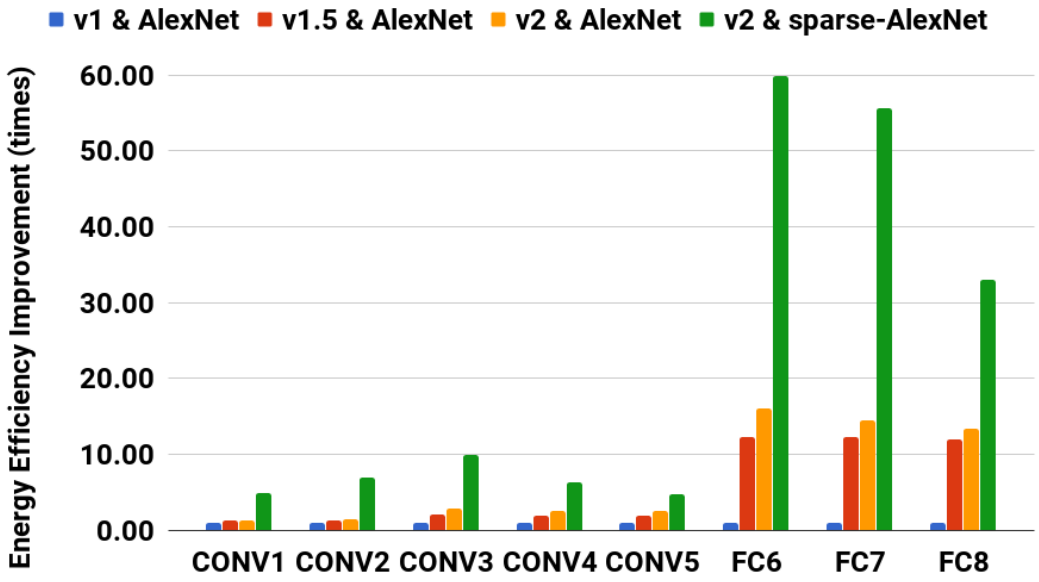


Figure 6-9: Energy efficiency improvement between different Eyeriss variants on AlexNet. Eyeriss v1 is used as the baseline for all layers. The experiment uses a batch size of 1.

performance slightly worse in the DW CONV layers. Overall, however, Eyeriss v2 is still 1.9× faster than Eyeriss v1.5, and is 10.9× faster than Eyeriss v1.

Fig. 6-11 shows the improvement on energy efficiency between different versions of Eyeriss on selected layers of MobileNet. While the energy efficiency of Eyeriss v1.5 is universally higher than Eyeriss v1 across all layers, it is again a mixed bag going from Eyeriss v1.5 to v2. Specifically, since the sparse PE does not benefit the processing of the DW CONV layers, the extra features in the sparse PE over the dense PE becomes overhead that reduce the energy efficiency of Eyeriss v2 as discussed in Section 6.5. Overall, Eyeriss v2 achieves a slightly better energy efficiency than Eyeriss v1.5, and is 1.9× more energy efficient than Eyeriss v1 on MobileNet. Note that the MobileNet used in this experiment is not sparse. We expect the performance of Eyeriss v2 will be even better if the MobileNet is pruned to improve sparsity as in the case of the sparse-AlexNet.

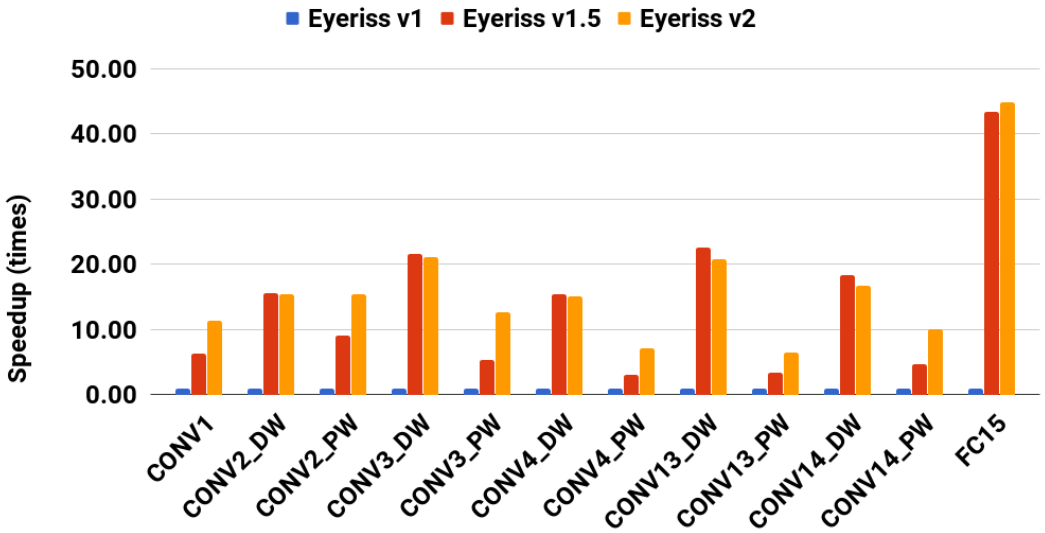


Figure 6-10: Performance speedup between different Eyeriss variants on MobileNet. Eyeriss v1 is used as the baseline for all layers. The experiment uses a batch size of 1.

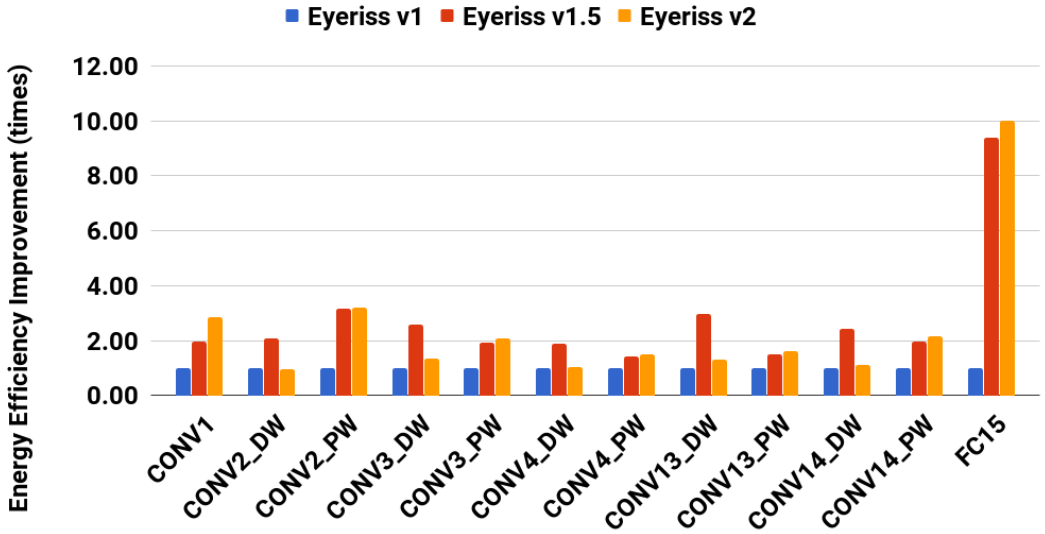


Figure 6-11: Energy efficiency improvement between different Eyeriss variants on MobileNet. Eyeriss v1 is used as the baseline for all layers. The experiment uses a batch size of 1.

6.7 Conclusions

Eyeriss v2 is an DNN accelerator architecture that supports the RS+ dataflow and features the hierarchical mesh network. It further improves the processing throughput and energy efficiency by exploiting (1) data sparsity in *both* weights and input activations and (2) SIMD processing. It can process sparse-AlexNet at 280.1 fps with an energy efficiency of 709.7 inferences/J. For MobileNet, which is a more compact DNN, it can achieve 1285.2 fps at 2020.8 inferences/J. We expect this result to further improve with pruned MobileNet that introduces more sparsity. Compared with Eyeriss v1, Eyeriss v2 achieves a speedup of 40$\times$ and 10 $\times$ with 11.3$\times$ and 1.9$\times$ higher energy efficiency on AlexNet and MobileNet, respectively.

Chapter 7

Conclusions and Future Work

7.1 Summary of Contributions

Research on the architectures for DNN accelerators is becoming popular for its promising performance and wide applicability. Despite the high volume of related work, this thesis has made the following unique contributions to the research community:

- Optimizing Dataflow for High Energy Efficiency: While data movement has already been recognized as the key to efficient processing of DNNs, **this thesis is the first one to describe how to systematically exploit a multi-level storage hierarchy with an optimized dataflow in the context of a spatial architecture for achieving superior energy efficiency**. Unlike the previous work that commonly applies one-size-fits-all dataflows regardless of the size and shape of the DNN, we have shown quantitatively that much better results could be achieved when the dataflow can adapt to the DNN data structure with the support of a re-configurable architecture.

- Optimizing PE Utilization for High Performance: The bottleneck in processing emerging DNNs is the high variation in data reuse, which can lead to reduced processing throughput due to the low utilization of PEs. **This thesis is also the first one to describe how to systematically improve the processing throughput through increasing both the number of active PEs and the percentage of active cycles for each PE**. Unlike the previous work that commonly uses looser performance bounds,

such as the peak performance or just the number of active PEs, to assess the hardware performance, we have quantitatively shown that the actual performance is often much lower when dealing with a diverse set of DNNs. This also motivates the need for a more flexible and adaptive architecture for data delivery.

- Frameworks for Energy Efficiency and Performance Analysis: The above analyses are only made possible thanks to our frameworks that can **systematically analyze and evaluate the energy efficiency and performance of different dataflows and micro-architectural designs working for different shapes and sizes of DNNs**. This should prove valuable for the development of future architectures. The approach of these frameworks is simple yet general, making it easy to fit into different architectural setups. In addition, the results are comprehensive, since it takes the entire system into account, instead of just the processor itself, which helps decision making during the design stage. The frameworks are also expandable and therefore future-proof. We expect these frameworks to be used by other researchers to explore architectural innovations, such as exploiting data compression and network sparsity, and quickly assess their impact at the system level.

- Global Design Optimization: This thesis will be referenced as evidence that neither minimizing the energy of the processor chip nor the DRAM alone is optimal. Instead, the evaluation has to consider the whole system at once. In addition, it shows that optimizing for all types of data (weights, feature maps, and partial sums) is important. Also, a system that cannot optimize for different shapes and sizes of the network models will see performance degradation. This knowledge not only helps future work to set the correct design objectives, but also urges them to consider the comparison using a set of relevant and complete metrics. We believe this effort can thus have a significant impact on the development of future architectures.

- A Taxonomy of Dataflows: Based on the knowledge learned from our analysis frameworks, we are able to identify the key differences between previous works that result in the most impact on energy efficiency and performance. This insight is condensed into a taxonomy that classifies related work into major categories based on their

dataflows. **This taxonomy is already gaining traction among researchers since it helps them to compare and contrast different designs despite differences in the lower-level details**. It greatly helps researchers within or even across disciplines to sift through the vast amount of publications by providing the big picture of the design. The categorical pros and cons for each dataflow can also be used as guidelines to improve future architectures.

- An Energy-Efficient Dataflow: The Row-Stationary (RS) dataflow implemented in Eyeriss v1 is the first proof-of-concept of a highly adaptive dataflow that optimizes for the system energy efficiency for deep neural network models. **Its high efficiency is verified by the measurement results on a fabricated test chip**.

- A Flexible Architecture for Data Delivery: To achieve a high utilization of PEs, we have proposed a combination of an improved RS+ dataflow and an adaptive hierarchical mesh NoC. The RS+ dataflow can parallelize the processing in any data dimension, which ensures a high number of active PEs; the adaptive NoC can provide enough bandwidth across a wide range of requirements without sacrificing data reuse, which ensures a high percentage of active cycles for each PE while maintaining high energy efficiency.

- A Milestone Domain-Specific Processor: **Eyeriss sets a milestone example of a domain-specific processor that serves a wide variety of DNNs in one simple design**. It achieves orders of magnitude higher throughput and energy efficiency than general-purpose processors while still being flexible enough to support a wide range of state-of-the-art deep neural networks. In addition, the Eyeriss architecture is realized by replicating low-complexity building blocks in a regular layout, which saves implementation cost.

Overall, this thesis demonstrates the importance of the co-design between the software and hardware architecture, in which dataflow is an important constituent, for the optimization of performance, energy efficiency and flexibility for DNN accelerators. Achieving this balance will open up more opportunities for AI to be applied in real-world applications with energy and performance constraints.

7.2 Future Work

As the research of DNNs is still moving at a very fast pace, many opportunities and challenges still lie ahead for the design of future DNN accelerators:

- *More Flexible Energy/Performance Modeling*: While we have proposed frameworks to analyze the energy efficiency and performance of DNN accelerators, oftentimes the challenge lies in finding the right trade-offs in a large design space. Currently the search for the optimal design is still a very manual process that requires insights from experienced designers. Taking into account the fast changing nature of popular DNN configurations, it is very hard for the designs of hardware architecture to keep up with how the applications evolve. A tool that can perform more flexible modeling on the energy and performance for the exploration of a large design space will be very crucial and is still an active research area.

- *Hardware-Friendly DNN Designs*: As pointed out above, currently the design of hardware architectures for DNN processing is still lagging behind the development of DNNs. This also creates a gap between what the algorithm designers perceive as hardware-friendly features and what the hardware architects would like to see being implemented in emerging DNNs. Research on how to provide instant feedback on the processing throughput and energy efficiency in the design or training of a DNN will become invaluable to bridge the gap and yield truly hardware-friendly DNNs. Some recent work in this direction can be found in [71, 72].

- *Hardware Architecture for Training*: This thesis mainly focuses on the inference part of DNN processing, while training is also an important aspect. Currently, training is usually done off-line due to its time-consuming nature and the high requirement on hardware resources. Building high performance and energy efficient hardware for training that can be applied across various compute platforms will open up even more opportunities, such as applications that emphasize customization and privacy.

- *Generalization to Other Applications*: Many design principles discovered and discussed in this thesis have the potential to be generalized to deal with challenges in

other fields. For example, many techniques proposed to deal with sparsity in DNNs can also benefit sparse linear algebra. Graph processing is another blooming area that can take inspirations from DNN accelerators since dataflow also plays a central role in its design.

Bibliography

[1] Jorge Albericio, Patrick Judd, Tayler Hetherington, Tor Aamodt, Natalie Enright Jerger, and Andreas Moshovos. Cnvlutin: ineffectual-neuron-free deep neural network computing. In *ISCA*, 2016.

[2] Doug Burger, Stephen W. Keckler, Kathryn S. McKinley, Mike Dahlin, Lizy K. John, Calvin Lin, Charles R. Moore, James Burrill, Robert G. McDonald, and William Yoder. Scaling to the End of Silicon with EDGE Architectures. *Computer*, 37(7), 2004.

[3] Lukas Cavigelli, David Gschwend, Christoph Mayer, Samuel Willi, Beat Muheim, and Luca Benini. Origami: A Convolutional Network Accelerator. In *GLSVLSI*, 2015.

[4] Srimat Chakradhar, Murugan Sankaradas, Venkata Jakkula, and Srihari Cadambi. A Dynamically Configurable Coprocessor for Convolutional Neural Networks. In *ISCA*, 2010.

[5] 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 *ASPLOS*, 2014.

[6] Yu-Hsin Chen. An 1000-class image classification task performed by the Eyeriss-integrated deep learning system. `https://vimeo.com/154012013`.

[7] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks. In *Proceedings of the 43rd Annual International Symposium on Computer Architecture (ISCA)*, 2016.

[8] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. Using Dataflow to Optimize Energy Efficiency of Deep Neural Network Accelerators. *IEEE Micro's Top Picks from the Computer Architecture Conferences*, 37(3), May-June 2017.

[9] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. Eyeriss v2: A Flexible and High-Performance Accelerator for Emerging Deep Neural Networks. In *submission to MICRO*, 2018.

[10] Yu-Hsin Chen, Tushar Krishna, Joel Emer, and Vivienne Sze. Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks. In *IEEE ISSCC*, 2016.

[11] Yu-Hsin Chen, Tushar Krishna, Joel Emer, and Vivienne Sze. Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks. *IEEE Journal of Solid-State Circuits*, 52:127–138, 2016.

[12] Yu-Hsin Chen, Tien-Ju Yang, Joel Emer, and Vivienne Sze. Understanding the limitations of existing energy-efficient design approaches for deep neural networks. In *SysML*, 2018.

[13] 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 *MICRO*, 2014.

[14] Bill Dally. Power, Programmability, and Granularity: The Challenges of ExaScale Computing. In *IEEE IPDPS*, 2011.

[15] Bhavya K. Daya, Chia-Hsin Owen Chen, Suvinay Subramanian, Woo-Cheol Kwon, Sunghyun Park, Tushar Krishna, Jim Holt, Anantha P. Chandrakasan, and Li-Shiuan Peh. SCORPIO: A 36-core Research Chip Demonstrating Snoopy Coherence on a Scalable Mesh NoC with In-network Ordering. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture (ISCA)*, pages 25–36, June 2014.

[16] Li Deng, Jinyu Li, Jui-Ting Huang, Kaisheng Yao, Dong Yu, Frank Seide, Michael Seltzer, Geoff Zweig, Xiaodong He, Jason Williams, et al. Recent advances in deep learning for speech research at Microsoft. In *ICASSP*, 2013.

[17] Richard Dorrance, Fengbo Ren, and Dejan Marković. A scalable sparse matrix-vector multiplication kernel for energy-efficient sparse-blas on FPGAs. In *ISFPGA*, 2014.

[18] 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 *ISCA*, 2015.

[19] Clément Farabet, Cyril Poulet, and Yann LeCun. An FPGA-based Stream Processor for Embedded Real-time Vision with Convolutional Networks. In *ICCV Workshop on Embedded Computer Vision*, 2009.

[20] Vinayak Gokhale, Jonghoon Jin, Aysegul Dundar, Berin Martini, and Eugenio Culurciello. A 240 G-ops/s Mobile Coprocessor for Deep Neural Networks. In *IEEE CVPRW*, 2014.

[21] David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1989.

[22] Venkatraman Govindaraju, Chen-Han Ho, and Karthikeyan Sankaralingam. Dynamically Specialized Datapaths for Energy Efficient Computing. In *IEEE HPCA*, 2011.

[23] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. Deep Learning with Limited Numerical Precision. *CoRR*, abs/1502.02551, 2015.

[24] 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 *ISCA*, 2016.

[25] Song Han, Huizi Mao, and William J. Dally. Deep Compression: Compressing Deep Neural Network with Pruning, Trained Quantization and Huffman Coding. In *ICLR*, 2016.

[26] Song Han, Jeff Pool, John Tran, and William Dally. Learning both Weights and Connections for Efficient Neural Network. In *Advances in Neural Information Processing Systems 28*, pages 1135–1143, 2015.

[27] John R. Hauser and John Wawrzynek. Garp: a MIPS processor with a reconfigurable coprocessor. In *IEEE FCCM*, 1997.

[28] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. In *IEEE CVPR*, 2016.

[29] Mark Horowitz. Computing's energy problem (and what we can do about it). In *IEEE ISSCC*, 2014.

[30] 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.

[31] Jason Howard, Saurabh Dighe, Yatin Hoskote, Sriram Vangal, David Finan, Gregory Ruhl, David Jenkins, Howard Wilson, Nitin Borkar, Gerhard Schrom, F. Pailet, S. Jain, T. Jacob, S. Yada, S. Marella, P. Salihundam, V. Erraguntla, M. Konow, M. Riepen, G. Droege, J. Lindemann, M. Gries, T. Apel, K. Henriss, T. Lund-Larsen, S. Steibl, S. Borkar, V. De, R. V. D. Wijngaart, and T. Mattson. A 48-Core IA-32 message-passing processor with DVFS in 45nm CMOS. In *Proceedings of IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, pages 108–109, Feb 2010.

[32] Forrest N. Iandola, Matthew W. Moskewicz, Khalid Ashraf, Song Han, William J. Dally, and Kurt Keutzer. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <1MB model size. *arXiv:1602.07360*, 2016.

[33] Natalie Enright Jerger and Li-Shiuan Peh. On-chip networks. *Synthesis Lectures on Computer Architecture*, 4(1):1–141, 2009.

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

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

[36] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. ImageNet Classification with Deep Convolutional Neural Networks. In *NIPS*, 2012.

[37] Hyoukjun Kwon, Ananda Samajdar, and Tushar Krishna. Rethinking nocs for spatial neural network accelerators. In *Proceedings of the Eleventh IEEE/ACM International Symposium on Networks-on-Chip*, page 19. ACM, 2017.

[38] 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*, pages 461–475. ACM, 2018.

[39] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, May 2015.

[40] Gideon Lewis-Kraus. The great A.I. Awakening. `https://www.nytimes.com/2016/12/14/magazine/the-great-ai-awakening.html`.

[41] Fei-Fei Li, Andrej Karpathy, and Justin Johnson. Stanford CS class CS231n: Convolutional Neural Networks for Visual Recognition. `http://cs231n.stanford.edu/`.

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

[43] Krishna T. Malladi, Benjamin C. Lee, Frank A. Nothaft, Christos Kozyrakis, Karthika Periyathambi, and Mark Horowitz. Towards energy-proportional datacenter memory with mobile dram. In *ISCA*, 2012.

[44] Bingfeng Mei, Serge Vernalde, Diederik Verkest, Hugo De Man, and Rudy Lauwereins. ADRES: An Architecture with Tightly Coupled VLIW Processor and Coarse-Grained Reconfigurable Matrix. In *FPL*, 2003.

[45] Cade Metz. Big Bets on A.I. Open a New Frontier for Chip Start-Ups, Too. `https://www.nytimes.com/2018/01/14/technology/artificial-intelligence-chip-start-ups.html`.

[46] Ethan Mirsky and Andre DeHon. MATRIX: a reconfigurable computing architecture with configurable instruction distribution and deployable resources. In *IEEE FCCM*, 1996.

[47] Bert Moons and Marian Verhelst. A 0.3–2.6 TOPS/W precision-scalable processor for real-time large-scale ConvNets. In *Symp. on VLSI*, 2016.

[48] Vinod Nair and Geoffrey E. Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th International Conference on International Conference on Machine Learning*, ICML'10, pages 807–814, USA, 2010. Omnipress.

[49] Tony Nowatzki, Vinay Gangadhar, and Karthikeyan Sankaralingam. Exploring the Potential of Heterogeneous Von Neumann/Dataflow Execution Models. In *ISCA*, 2015.

[50] Nvidia. NVDLA Open Source Project, 2017.

[51] Angshuman Parashar, Michael Pellauer, Michael Adler, Bushra Ahsan, Neal Crago, Daniel Lustig, Vladimir Pavlov, Antonia Zhai, Mohit Gambhir, Aamer Jaleel, Randy Allmon, Rachid Rayess, Stephen Maresh, and Joel Emer. Triggered Instructions: A Control Paradigm for Spatially-programmed Architectures. In *ISCA*, 2013.

[52] 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 43rd Annual International Symposium on Computer Architecture (ISCA)*, 2017.

[53] Seongwook Park, Kyeongryeol Bong, Dongjoo Shin, Jinmook Lee, Sungpill Choi, and Hoi-Jun Yoo. A 1.93TOPS/W scalable deep learning/inference processor with tetra-parallel MIMD architecture for big-data applications. In *IEEE ISSCC*, 2015.

[54] Maurice Peemen, Arnaud A. A. Setio, Bart Mesman, and Henk Corporaal. Memory-centric accelerator design for Convolutional Neural Networks. In *IEEE ICCD*, 2013.

[55] Wajahat Qadeer, Rehan Hameed, Ofer Shacham, Preethi Venkatesan, Christos Kozyrakis, and Mark A. Horowitz. Convolution Engine: Balancing Efficiency and Flexibility in Specialized Computing. In *ISCA*, 2013.

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

[57] Murugan Sankaradas, Venkata Jakkula, Srihari Cadambi, Srimat Chakradhar, Igor Durdanovic, Eric Cosatto, and Hans P. Graf. A Massively Parallel Coprocessor for Convolutional Neural Networks. In *IEEE ASAP*, 2009.

[58] Herman Schmit, David Whelihan, Andrew Tsai, Matthew Moe, Benjamin Levine, and R. Reed Taylor. PipeRench: A virtualized programmable datapath in 0.18 micron technology. In *IEEE CICC*, 2002.

[59] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel,

and Demis Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489, Jan. 2016.

[60] Jaehyeong Sim, Jun-Seok Park, Minhye Kim, Dongmyung Bae, Yeongjae Choi, and Lee-Sup Kim. A 1.42TOPS/W deep convolutional neural network recognition processor for intelligent IoE systems. In *2016 IEEE International Solid-State Circuits Conference (ISSCC)*, pages 264–265, Jan 2016.

[61] Karen Simonyan and Andrew Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition. *CoRR*, abs/1409.1556, 2014.

[62] Vinay Sriram, David Cox, Kuen Hung Tsoi, and Wayne Luk. Towards an embedded biologically-inspired machine vision processor. In *FPT*, 2010.

[63] Steven Swanson, Andrew Schwerin, Martha Mercaldi, Andrew Petersen, Andrew Putnam, Ken Michelson, Mark Oskin, and Susan J. Eggers. The WaveScalar Architecture. *ACM TOCS*, 25(2), 2007.

[64] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel S. Emer. Efficient processing of deep neural networks: A tutorial and survey. *Proceedings of the IEEE*, 105(12):2295–2329, Dec 2017.

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

[66] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. In *CVPR*, 2016.

[67] Jesmin Jahan Tithi, Neal C. Crago, and Joel S. Emer. Exploiting spatial architectures for edit distance algorithms. *IEEE ISPASS*, 2014.

[68] Fengbin Tu, Shouyi Yin, Peng Ouyang, Shibin Tang, Leibo Liu, and Shaojun Wei. Deep convolutional neural network architecture with reconfigurable computation patterns. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 25(8):2220–2233, 2017.

[69] R. Clint Whaley and Jack J. Dongarra. Automatically tuned linear algebra software. In *Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*, SC '98, pages 1–27, Washington, DC, USA, 1998. IEEE Computer Society.

[70] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, Apr 2009.

[71] Tien-Ju Yang, Yu-Hsin Chen, and Vivienne Sze. Designing Energy-Efficient Convolutional Neural Networks using Energy-Aware Pruning. In *CVPR*, 2017.

[72] Tien-Ju Yang, Andrew Howard, Bo Chen, Xiao Zhang, Alec Go, Vivienne Sze, and Hartwig Adam. Netadapt: Platform-aware neural network adaptation for mobile applications. *arXiv preprint arXiv:1804.03230*, 2018.

[73] Shouyi Yin, Peng Ouyang, Shibin Tang, Fengbin Tu, Xiudong Li, Leibo Liu, and Shaojun Wei. A 1.06-to-5.09 tops/w reconfigurable hybrid-neural-network processor for deep learning applications. In *VLSI Circuits, 2017 Symposium on*, pages C26–C27. IEEE, 2017.

[74] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks. In *FPGA*, 2015.