GENERAL REUSE-CENTRIC CNN ACCELERATOR

A THESIS SUBMITTED TO

THE GRADUATE SCHOOL OF ENGINEERING AND SCIENCE

OF BILKENT UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR

THE DEGREE OF

MASTER OF SCIENCE

IN

COMPUTER ENGINEERING

By

Nihat Mert ÇİÇEK

February 2021

GENERAL REUSE-CENTRIC CNN ACCELERATOR
By Nihat Mert ÇİÇEK
February 2021

We certify that we have read this thesis and that in our opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Özcan ÖZTÜRK(Advisor)

Uğur GÜDÜKBAY

Süleyman TOSUN

Approved for the Graduate School of Engineering and Science:

Ezhan KARAŞAN
Director of the Graduate School

ABSTRACT

GENERAL REUSE-CENTRIC CNN ACCELERATOR

Nihat Mert ÇİÇEK
M.S. in Computer Engineering
Advisor: Özcan ÖZTÜRK
February 2021

Reuse-centric CNN acceleration speeds up CNN inference by reusing computations for similar neuron vectors in CNN's input layer or activation maps. This new paradigm of optimizations is however largely limited by the overheads in neuron vector similarity detection, an important step in reuse-centric CNN. This thesis presents the first in-depth exploration of architectural support for reuse-centric CNN. It proposes a hardware accelerator, which improves neuron vector similarity detection and reduces the energy consumption of reuse-centric CNN inference. The accelerator is implemented to support a wide variety of network settings with a banked memory subsystem. Design exploration is performed through RTL simulation and synthesis on an FPGA platform. When integrated into Eyeriss, the accelerator can potentially provide improvements up to 7.75X in performance. Furthermore, it can make the similarity detection up to 95.46% more energy-efficient, and it can accelerate the convolutional layer up to 3.63X compared to the software-based implementation running on the CPU.

Keywords: CNN, reuse-centric, accelerator, inference.

ÖZET

GENEL YENİDEN KULLANIM MERKEZLİ CNN HIZLANDIRICI

Nihat Mert ÇİÇEK
Bilgisayar Mühendisliği, Yüksek Lisans
Tez Danışmanı: Özcan ÖZTÜRK
Şubat 2021

Yeniden kullanım merkezli CNN hızlandırma yöntemi, CNN'nin giriş katmanında veya aktivasyon haritalarında benzer nöron vektörleri için hesaplamaları yeniden kullanarak CNN çıkarımını hızlandırır. Ancak bu yeni optimizasyon paradigması, yeniden kullanım merkezli CNN'de önemli bir adım olan nöron vektör benzerliği tespitindeki genel giderlerle büyük ölçüde sınırlıdır. Bu tez, yeniden kullanım merkezli CNN için mimari destek üzerine ilk derinlemesine araştırmayı sunmaktadır. Nöron vektör benzerlik tespitini geliştiren ve yeniden kullanım merkezli CNN çıkarımının enerji tüketimini azaltan bir donanım hızlandırıcı önerir. Hızlandırıcı, bankalı bellek alt sistemi ile birlikte çok çeşitli ağ ayarlarını destekleyecek şekilde gerçeklenir. Tasarım keşfi, RTL simülasyonu ve bir FPGA platformunda sentez yoluyla gerçekleştirilir. Eyeriss'e entegre edildiğinde, hızlandırıcı potansiyel olarak performansta 7.75 kata kadar iyileştirmeler sağlayabilir. Ayrıca işlemci üzerinde koşan yazılım tabanlı gerçekleme ile karşılaştırıldığında, benzerlik tespitini %95.46'ya kadar daha enerji verimli ve evrişim katmanını 3.63 kata kadar daha hızlı yapabilir.

Anahtar sözcükler: CNN, donanım ivmelendirici/hızlandırıcı, veriyi yeniden kullanım, çıkarım.

Acknowledgement

This work has been supported in part by a grant from the Presidency of Defence Industries (SSB) and Aselsan A.Ş. with the researcher training program for defence industry (SAYP).

I would like to thank my thesis advisor Dr. Özcan Öztürk for his support throughout my master's work.

Next, I would like to thank Dr. Xipeng Shen and his doctoral student Lin Ning for their valuable collaboration and their great efforts and contributions to implementing the software related work.

Besides, I am grateful to my colleagues Çağla Irmak Rumelili Köksal, Mehmet Ali Gülden, and Dr. Fatih Say for all their support.

Finally, I would like to thank my family, who always supported me.

Contents

1 Introduction **1**

2 Background **4**

3 Overall Design Considerations **7**

4 Basic Design of General Reuse Discovery Engine **11**

4.1 Similarity Detection Process 11

4.2 Base Architecture 14

4.2.1 Reduction Unit (RU) 15

4.2.2 Load-Store-Execute Unit (LSEU) 15

4.2.3 Controller 16

4.3 Limitations 16

5 Advanced Accelerator Design **18**

5.1 Fetch Unit (FU) 19

5.2 Neuron Vector Table (NVT) 21
5.3 Reduction Unit (RU) 22
5.4 Dispatch Unit (DU) 25
5.5 Load-Store-Execute Unit (LSEU) 26
5.6 Controller 28
5.7 Discussion 28
5.7.1 Folding Support in Fetch Unit 28
5.7.2 Unfolding Engine 30
5.8 Synergy with Other CNN Accelerators 31

6 Evaluation 33

6.1 Experimental Setup 33
6.1.1 Tools 33
6.1.2 Benchmarks 34
6.1.3 Data Sets 34
6.1.4 Default Parameters 34
6.1.5 Execution Environments 35
6.2 Performance and Synergy with Existing Accelerators 37
6.3 Energy Reduction 40

6.4 Area Breakdown and Resource Utilization 40

6.5 Discussion 42

7 Related Work **44**

8 Conclusion **47**

List of Figures

2.1 An example matrix multiplication based convolution operation leveraging general reuse. Color coding shows the available reuse. . . 5

3.1 Matrix multiplication based general reuse-centric convolution operation (similar to the one implemented in [1]). 8

4.1 Illustration of our accelerator in a similarity detection module. . . 12

4.2 Base reuse-centric accelerator architecture. 14

5.1 Advanced accelerator design with fetch unit, reduction unit, dispatch unit, load-store-execute unit, and neuron vector table. . . . 19

5.2 An example fetch unit execution. 20

5.3 Banked content addressable memory and neuron vector allocation. 21

5.4 Details of the reduction unit with four modules, namely dispatch, execute, collector, and index. 23

5.5 Forwarding adder network topology with collector and index modules. 24

5.6 Example execution scenario for the dispatch unit with N collectors and M banks. 25

5.7 The load-store-execute unit (LSEU) is composed of cluster memories, ID-Key memory, and centroid execution unit 26

5.8 Fetch unit supporting fold operation. 29

5.9 Advanced architecture supporting on the fly unfolding operation. . 30

5.10 Top level architecture for the integration of the reuse-centric accelerator into Eyeriss [2]. 31

6.1 Average performance improvement over mobile CPU obtained through basic and advanced design for the similarity discovery module. 37

6.2 Performance improvement of reuse-centric implementations on CPU and with the proposed accelerator. The baseline is the default convolution performed on a mobile CPU. 38

6.3 The performance improvement brought to Eyeriss by our reuse-centric accelerator. 39

6.4 Energy reduction over mobile CPU through advanced design. . . . 40

6.5 Area breakdown of the modules in the advanced architecture based on the lookup tables (LUTs) and registers. 41

List of Tables

1.1 The average cluster sizes of the convolutional layers in CifarNet, ImageNet, and VGG-19 that keep the original prediction accuracy. 2

3.1 The time breakdown of the general reuse-centric convolution operation. S.D. is similarity detection. C.MM is centroid matrix-multiplication. F.D. is full-result derivation [1]. 9

6.1 Convolutional network and dataset pairs used in evaluation. . . . 34

6.2 Default parameters used for the advanced architecture. 35

6.3 Algorithm specific parameters used for the benchmarks. 36

6.4 Resources used by each module of the advanced design implemented on VCU118 board. 41

6.5 Comparison of theoretical size with actual size for cluster memory. 43

Chapter 1

Introduction

Convolutional Neural Networks (CNN) have been widely used in many data mining and machine learning domains in recent years. They are the pillars supporting many important tasks, from object recognition to autonomous driving, gesture recognition, and so on. The inference speed and energy efficiency of CNN are essential for it to work effectively on embedded devices.

Prior efforts on optimizing CNN inference efficiency largely fall into three categories. Some explore different convolution algorithms [3, 4], some compress networks [5], and most of all optimize matrix multiplications, the core operation in CNN, which mainly resorts to the leverage of network sparsity [6, 7], memory layout [8], and special hardware units [9, 10, 11, 12].

As a complementary aspect to these efforts, reusing similar neuron vectors (sequence of consecutive neurons in a convolutional layer) have been explored in the literature [13, 14]. There exists a wide set of neuron vector similarities in popular image datasets as shown in Table 1.1. As can be seen from this table, the three networks can maintain the default inference accuracy if similar neuron vectors (lengths vary across layers) are grouped together and only the cluster centers are used in the CNN computations. The average cluster sizes are as large as 6–100 neuron vectors.

Despite the potential of this method, discovering similar neurons takes longer than the original matrix multiplication and their software implementation suffers from increased execution time and energy consumption, as reported in [13, 14]. This is due to the fact that software implementation running on a general-purpose CPU with a specific instruction set executes on vectors in a serial fashion, which is compute-intensive. Moreover, inefficient use of caches cause increased energy consumption. A dedicated hardware solution can drastically reduce both execution time and energy consumption by optimizing memory accesses and parallelizing the required operations.

Table 1.1: The average cluster sizes of the convolutional layers in CifarNet, ImageNet, and VGG-19 that keep the original prediction accuracy.

Network	**DataSet**	**AVG Cluster Size**
CifarNet	Cifar10	100
AlexNet	ImageNet	6
VGG	ImageNet	8

In this work, we propose *general reuse-centric CNN accelerator*, which is a special hardware engine designed to accelerate convolution layers in CNNs. Our key contributions are the following:

(1) In order to leverage general reuse opportunities, we design a similarity detection engine implementing locality sensitive hashing (LSH) [15, 16] algorithm.

(2) Uninterrupted data flow, parallel data processing, and storage as well as resource efficiency are critical to achieving high performance and energy efficiency for inference. Therefore, we propose a superscalar nonblocking hardware pipeline, assisted with customized static random access memories (SRAMs) and content-addressable memories (CAMs).

(3) CNNs consist of many convolutional layers with different sizes and shapes. Thus, during inference, fast online adaptation to different layer configurations such as neuron vector lengths or image/activation map size is crucial. Towards this goal, we support the online reconfiguration of all units in our accelerator through a register mapped interface.

(4) Our accelerator supports integration into existing accelerators. Acting as a pre-processing unit, it reduces the total number of computations during the execution of convolutional layers of the target accelerator.

In the rest of this thesis, we first briefly introduce the background and outline the overall design considerations in Chapter 2 and Chapter 3, respectively. Next, we describe the basic design of the *general reuse discovery engine* in Chapter 4. The advanced features of the accelerator in addition to how it can be integrated into existing software and hardware CNN implementations are given in Chapter 5. We discuss our experimental evaluation in Chapter 6, and finally we conclude.

Chapter 2

Background

In this chapter, we give the basics of convolutional neural networks, an example of general reuse, and a brief explanation for hardware acceleration.

Convolutional Neural Networks (CNNs)

Neural networks, a subfield of artificial intelligence, perform a combination of linear and nonlinear functions. These functions are implemented through multiple layers, each fulfilling a special algorithm. When a convolution operation is included in a layer, it is called a convolutional neural network (CNN). Besides, they get the prefix "deep" when consisting of many layers.

For a specific task, the parameters of the functions in each layer of a CNN are optimized through a process called training. After the training step, running the network for accomplishing the task is called inference. A CNN is generally trained once on a powerful compute node such as GPU or TPU. On the other hand, the inference is performed many times on the edge device. Therefore, it is critical to perform inference with high performance and energy efficiency since the target device usually has limited resources.

A typical CNN as a deep neural network consists of many layers, including the convolutional layer, the nonlinear layer, the pooling layer, the normalization layer,

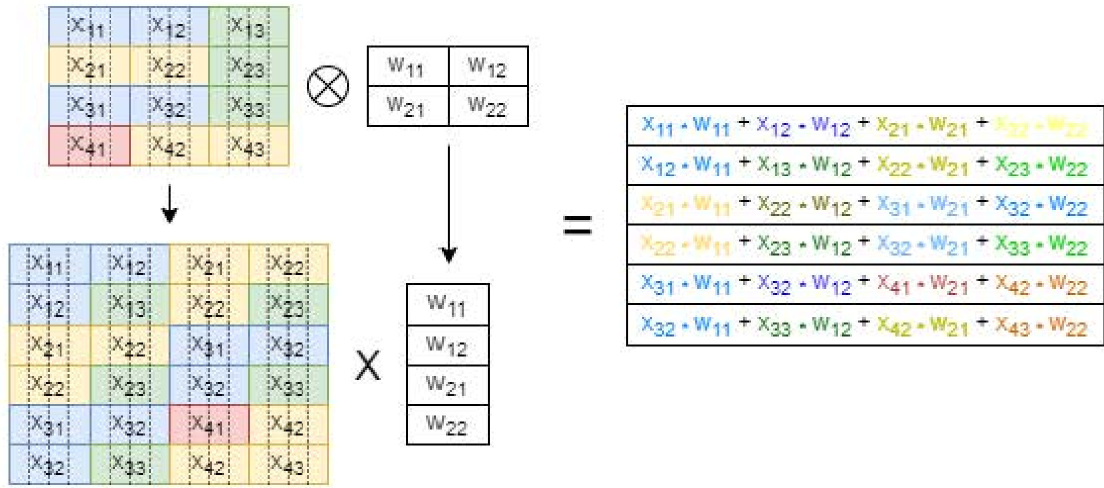


Figure 2.1: An example matrix multiplication based convolution operation leveraging general reuse. Color coding shows the available reuse.

and the fully connected layer [17]. Among them, the most time consuming one is the convolutional layer [18]. Therefore, many researchers focus on accelerating the convolutional layer during inference [6, 7, 9, 12, 2].

A widely used method to implement convolution, the *matrix multiplication-based method*, unfolds the inputs into matrices and applies general matrix multiplication for computation. An example of a one-dimensional convolution using this method is illustrated in Figure 2.1. In this figure, an input image is unfolded into a large input matrix $\mathbf{X}$ for zero strides and padding. Then, it is multiplied with an unfolded weight matrix $\mathbf{W}$, producing an output matrix $\mathbf{y}$ such that $\mathbf{y} = \mathbf{X} \cdot \mathbf{W}$. It is important to note that a major inefficiency of the matrix multiplication-based implementation is additional memory requirement due to unfolding the input (input image or activation maps) into a large input matrix.

General Reuse

As can be seen in Figure 2.1, *general reuse* saves computations in CNN. More specifically, in this figure, $\mathbf{X}$ is an unfolded input matrix where each element

corresponds to the value of one neuron. Every four elements in the example correspond to the value of a neuron vector. Based on neuron vector similarities, the twenty-four neuron vectors are divided to twelve groups, represented in different colors. The dot product with a weight vector (e.g., $\mathbf{x}_{11} \cdot \mathbf{w}_{11}$) can be reused for the neuron vectors in the same group (e.g., $\mathbf{x}_{12} \cdot \mathbf{w}_{11}$, $\mathbf{x}_{31} \cdot \mathbf{w}_{11}$, and $\mathbf{x}_{32} \cdot \mathbf{w}_{11}$).

Hardware Acceleration

In DNN processing, most of the clock cycles and energy is spent during memory (DRAM) read or write operations. High performance and energy efficiency can be provided by designing a multiple level memory hierarchy, similar to multilevel cache structure in modern processors, or using buffers with different sizes. To minimize the cost, it is necessary to decrease the number of accesses to higher levels and to increase the reuse of data in lower levels. For this purpose, there exist special architectures called hardware accelerators [19] that optimize dataflow between memory and processing engines (PE). In these architectures, each processing engine can have its local memory and control logic to increase the reuse of data inside/across PEs so that computation and energy saving can be obtained.

Chapter 3

Overall Design Considerations

In this chapter, we will first describe general reuse-centric convolution operation. Then, we will present key requirements for strong acceleration and associate them with our design.

General Reuse-Centric CNN Acceleration and Performance Bottleneck

Matrix multiplication based general reuse-centric convolution operation consists of several stages as shown in Figure 3.1 [1]. First, the input image or activation map is converted to matrix form according to the shape of the weight matrix, stride, and padding settings of the convolutional layer. Using this matrix, similar neuron vectors are identified and cluster groups are formed. Then, matrix multiplication is performed between cluster centroids and the weight matrix. Finally, convolution output is derived by replicating partial results corresponding to the original neuron vectors.

The fraction of execution times of each component for a software-based general reuse-centric implementation running on a general-purpose CPU is given in Table 3.1. The similarity detection module takes 44% - 52% of the total execution time for CifarNet and 18% - 41% of total runtime for AlexNet and thus is the bottleneck of this acceleration method [1].

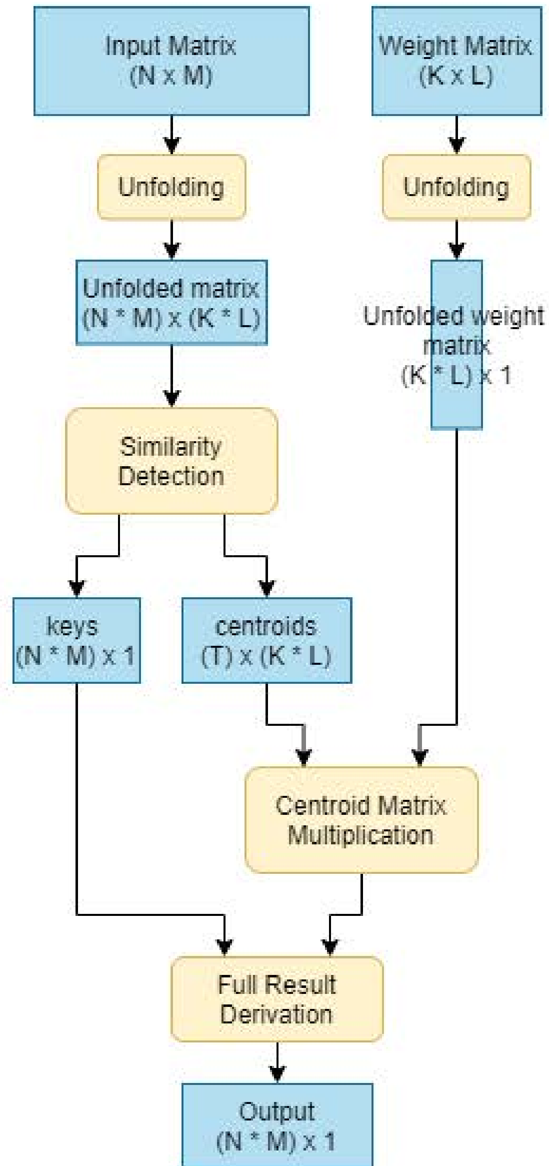


Figure 3.1: Matrix multiplication based general reuse-centric convolution operation (similar to the one implemented in [1]).

Table 3.1: The time breakdown of the general reuse-centric convolution operation. S.D. is similarity detection. C.MM is centroid matrix-multiplication. F.D. is full-result derivation [1].

Network	Layer	Unfold	S.D.	C.MM	F.D.
CifarNet	conv1	16.65%	52.56%	3.63%	3.84%
	conv2	4.61%	44.33%	17.30%	33.76%
AlexNet	conv1	10.80%	41.85%	24.18%	23.17%
	conv2	1.33%	27.08%	54.72%	23.17%
	conv3	0.70%	15.68%	47.52%	36.11%
	conv4	0.67%	18.97%	65.18%	15.18%
	conv5	1.01%	22.91%	47.48%	28.60%

Key Requirements

The following principles are taken into consideration throughout the design for high performance and low energy consumption.

- *Scalability:* The accelerator needs to scale well in terms of area and cost for being able to be integrated into various hardware accelerators. This also requires using computational and memory resources efficiently.
- *Flexibility:* The accelerator should be able to support different image sizes or activation maps.
- *Efficiency:* The accelerator should focus on utilizing all resources effectively.

Hardware Accelerator Design

Based on the percentages given in Table 3.1, the most time-consuming blocks are similarity detection and matrix multiplication. Since many existing accelerators already have efficient matrix multiplication stages, we focus on leverage similarity detection. This way, we aim to increase performance and decrease the energy consumption of existing accelerators by integrating this engine.

Our design has several parameters in order to configure it for different kinds of accelerators, ranging from CPUs to GPUs and TPUs, aligning well with the first

principle, scalability. In addition, our accelerator adapts fast to differences in layers and network parameters meeting the second principle, flexibility. Furthermore, it utilizes all available resources efficiently by executing different neuron vectors in parallel. Besides, it has customized SRAMs to keep generated cluster centroids and keys efficiently, thereby, achieving efficiency.

Chapter 4

Basic Design of General Reuse Discovery Engine

This work tries to reduce the performance bottleneck in *neuron vector similarity identification* with a hardware-software co-design approach. More specifically, we focus on accelerating the convolution layers through a hardware accelerator and a similarity detection method. This chapter starts with an explanation of the details of the operations involved in *similarity detection* and then describes the base architecture of the accelerator.

4.1 Similarity Detection Process

This section describes the design details of the similarity detection process required to perform convolution operation.

Similarity discovery can be applied to the entire input matrix or to smaller input submatrices. As specified in [13], the clustering scope determines which of the two options is more effective. When clustering is performed within a single

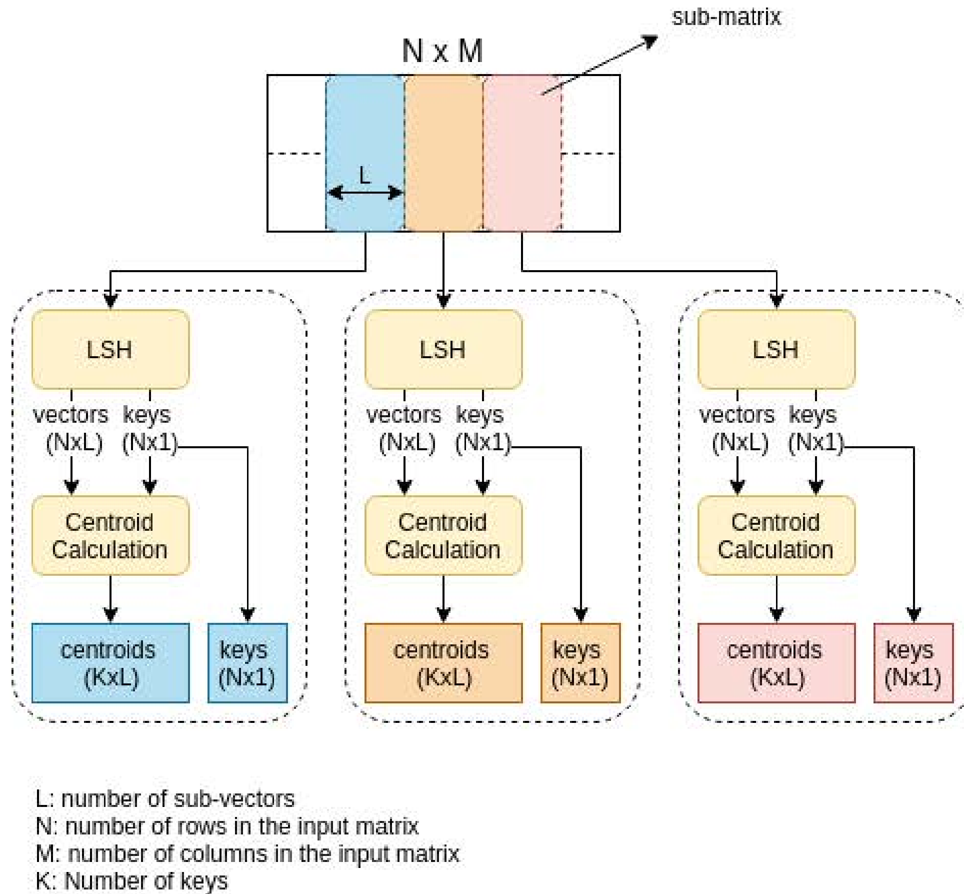


Figure 4.1: Illustration of our accelerator in a similarity detection module.

input matrix, namely for a small scope, whole-vector clustering is better than sub-vector clustering. However, when clustering is performed across batches, namely for a large scope, sub-vector clustering generally beats the other. In addition, when we look at it from the hardware perspective, using sub-vector clustering enables us to operate in parallel via a multi-accelerator architecture. Therefore, in this work, we choose sub-vector clustering as the granularity. Besides, it is important to note that one more step is required to get overall convolution output in sub-vector clustering: partial sum accumulation after centroid matrix multiplication.

For similarity detection, it is necessary to choose a clustering method that is hardware-friendly and does not degrade the original accuracy of the network.

In literature, there are different types of clustering methods to find similar neuron vectors. Three clustering algorithms including K-means, Hyper-Cube, and Locality-Sensitive Hashing (LSH) are explored in [13] for similarity detection. According to this study, although the Hyper-Cube algorithm is fast and efficient, it fails for large neuron vectors. Another clustering algorithm K-means gives good clustering results; however, its clustering overhead is larger than the original matrix multiplication, thus making it impractical. An alternative to these methods, LSH based similarity detection, offers both high accuracy and low computation overhead. Therefore, we choose LSH as the clustering method for discovering similarities among neuron vectors.

LSH algorithm mainly performs the dot product between each neuron vector and randomly filled hashing vectors in order to extract similarities [20, 16, 21, 22, 15]. For a given H hashing vector, each neuron vector goes into H dot products, resulting in H vectors. When sign bits of all H vectors are concatenated, a cluster key is obtained. After this operation, generated cluster keys with original neuron vectors are used to calculate cluster centroids. More specifically, the arithmetic mean of all neuron vectors in the same cluster represents the centroid for that cluster.

The whole similarity detection process is illustrated in Figure 4.1. For an input matrix of size NxM and for sub-vector size L, we have $S = M/L$ subvectors. Each sub-vector goes into the LSH unit and centroid calculation unit, generating cluster centroids and keys.

In this section, we represented a generic framework for similarity detection. We will explain how we implement LSH based similarity detection in hardware in the following section.

4.2 Base Architecture

In this section, we propose a base architecture whose main objective is to find the similarities among neuron vectors. This architecture includes three modules as reduction unit (RU), load-store-execute unit (LSEU), and controller. The overall architectural block diagram is given in Figure 4.2.

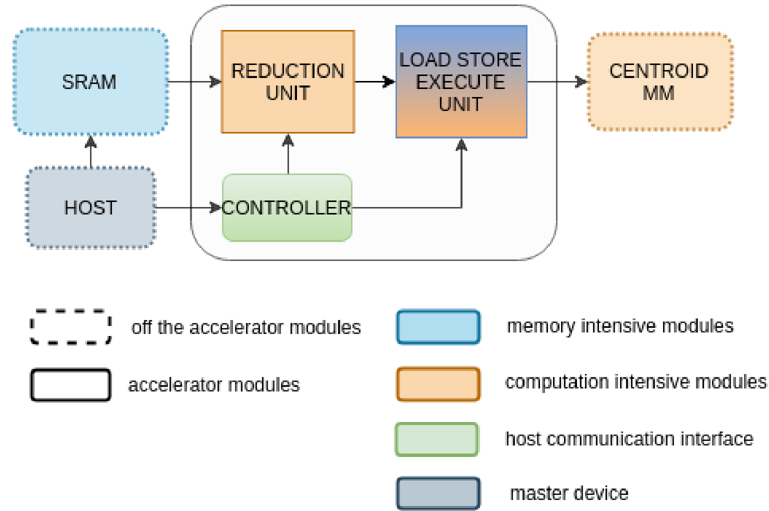


Figure 4.2: Base reuse-centric accelerator architecture.

The host starts the execution by interacting with the controller inside the accelerator. Then, the controller issues signal required to control overall data flow between host, execution units (RU and LSEU), and memory interface (SRAM). After the accelerator is configured by the host for the convolutional layer's specific parameters, vectors start streaming into the reduction unit for dot product calculation of the LSH algorithm. At the end of this stage, cluster keys are generated and fed to LSEU for updating the cluster centroid. The details of each block are described in the following subsections.

4.2.1 Reduction Unit (RU)

This module mainly computes the dot product of neuron vectors with hash tables and generates the cluster key. Dot product operation involves a series of multiplication and addition. Therefore, adders and multipliers are placed and routed based on a tree-shaped architecture in a nonblocking pipelined fashion. First, fetched neuron vectors from SRAM goes into multiplication with hash tables. Then, the results of multiplications are fed to the reduction tree, generating dot product results. At the end of this stage, cluster keys are generated using the sign of the dot product result as described in Section 4.1.

Furthermore, it can be reconfigured online by the controller to support the execution of different kinds of layers and networks. Simply, idle adders and multipliers are disabled during the execution.

4.2.2 Load-Store-Execute Unit (LSEU)

This unit calculates and updates the cluster centroid for newly generated neuron vectors. Generated cluster keys from the reduction unit are used to load the latest centroid value from memory. Using the latest centroid information and new neuron vector, it performs operations required to get the arithmetic average of all neuron vectors inside the cluster. Then, generated new cluster centroid is written back to the memory. In addition, each neuron vector's cluster key is stored in memory to be used later by the centroid matrix multiplication module.

In this implementation, we use SRAM as the storage element. There are three different sized SRAMs in this unit. First, *Cluster Centroid SRAM* keeps the up-to-date centroid information for each valid cluster. Second, *ID SRAM* keeps the cluster key of each neuron vector. Third, valid clusters are kept in *Valid SRAM*.

4.2.3 Controller

The controller unit serves as the communication interface of the accelerator with the outside world. The host can access and configure the accelerator through this interface. In addition, the functional accuracy of the operation performed is guaranteed by this module. For example, when there are multiple neuron vectors with the same cluster key along the pipeline, the controller can stall the pipeline or forward the up-to-date centroid to ensure correctness.

In our design, the controller has a register space for the host to read and write. This way, the host can start the execution, set the network and layer-related parameters, read the cluster centroids and key information for each neuron vector processed and send them to the centroid matrix multiplication module in order to complete the convolution operation. Due to its simple read-write interface, it can be integrated into any host interface easily.

4.3 Limitations

Limitations in the base architecture are twofold, namely computation and memory.

Different convolutional layers have different properties in terms of neuron vector size and hash size. Therefore, our accelerator must support the online reconfiguration of the network-related parameters. Current base architecture implementation suffers from efficient use of hardware resources during this reconfiguration. For example, the reduction unit can process only one neuron vector at a time. Depending on the neuron vector size setting, some computational units may stay idle during the execution. On the other hand, for small-sized neuron vectors, it is possible to calculate the dot product of more than one neuron vector in one cycle. This way, resources are utilized more efficiently and performance is increased greatly by parallel processing of neuron vectors, following the second and third key requirement described in Chapter 3, flexibility and efficiency, respectively.

For this purpose, a fetch unit can be added to the design, as will be described in Chapter 5. Besides, the reduction unit will be improved by adding the parallel processing capability of different neuron vectors.

Memory is also a critical component in terms of performance and energy. As memory size gets larger, access time becomes longer, and energy consumption increases. Therefore, performance and energy requirements put a limit on the maximum memory size supported. Furthermore, even implementing a large memory may not be possible because of technological limitations. Since we are limited in terms of memory size, we need to use it efficiently. However, in this base architecture implementation, the memory may not be utilized due to different neuron vector sizes and layer configurations. For example, a layer with a large hash size and a small neuron vector size requires a memory that has big address width and small data width. On the other hand, another layer may just have the opposite. Thus, it is necessary to use available memory effectively for any kind of network, meeting the second and third principles described in Chapter 3, flexibility and efficiency, respectively. As a result, we focus on efficient use of memory in the advanced architecture by designing a special dispatch unit and load-store-execute unit.

Chapter 5

Advanced Accelerator Design

As discussed earlier, base architecture does not have the dynamic reconfiguration feature and is not able to process and store different neuron vectors in parallel; thus limiting the performance and energy. By designing a special fetch unit, reduction unit, dispatch unit, custom static random access memories (SRAM), and content-addressable memories (CAM) with banked memory support, we aim to overcome those limitations. High-level block diagram of our advanced architecture is given in Figure 5.1. In the rest of this chapter, we will explain each unit in detail.

As can be seen from this figure, the overall design consists of several stages. In the first stage, the fetch unit brings in data from SRAM and processes it to create neuron vectors for the given layer. Then, it checks sparsity for each neuron vector and removes the ones with a zero value, which saves energy and resources greatly. Besides, the neuron vector is saved into a CAM to prevent unnecessary data flow along the pipeline. Afterward, neuron vectors are fed into a reduction tree, performing dot product operation and generating cluster keys. In the next step, the bank number is determined for ready keys and they are dispatched to the load-store-execute unit (LSEU). This unit fetches a previously-stored neuron vector from CAM and updates the centroid for the generated cluster key. Finally, it stores cluster key information for each identity and it serves the host for centroid matrix multiplication and full result derivation stage mentioned in Chapter 3.

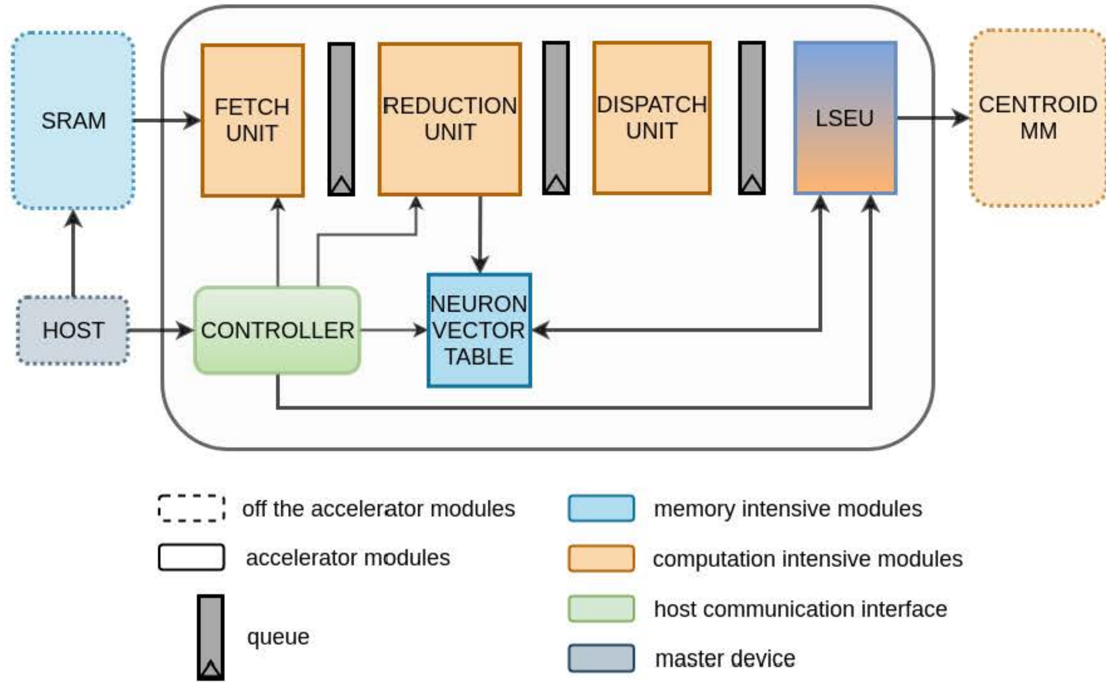


Figure 5.1: Advanced accelerator design with fetch unit, reduction unit, dispatch unit, load-store-execute unit, and neuron vector table.

5.1 Fetch Unit (FU)

The fetch module consists of four submodules, namely, identity generator, sparsity controller, cam allocator, and fetch target queue, connected in a pipelined fashion.

The first submodule, identity generator, processes the raw data for various vector sizes and SRAM row width settings. This flexibility brings us two opportunities: 1) compatibility with any image size or layer size 2) ease of integration into any existing accelerator. In addition, it is also capable of processing more than one neuron vector in a single cycle. This way, it is possible to achieve parallel processing and storage, thereby, increasing performance greatly.

The output of the identity pattern generator is fed into the sparsity controller. This sub-module detects and removes zeros from each neuron vector since there is no need to perform a reduction operation. This way, energy is greatly saved for large sparse image and activation matrices.

After sparsity check, a new entry is allocated at the neuron vector table for each newly generated identity by the CAM allocator sub-module. Neuron vectors are stored in CAM for later use by the load-store-execute unit. Here, it is important to note that only one entry should be allocated for each identity. In our design, it satisfies this condition for any vector size.

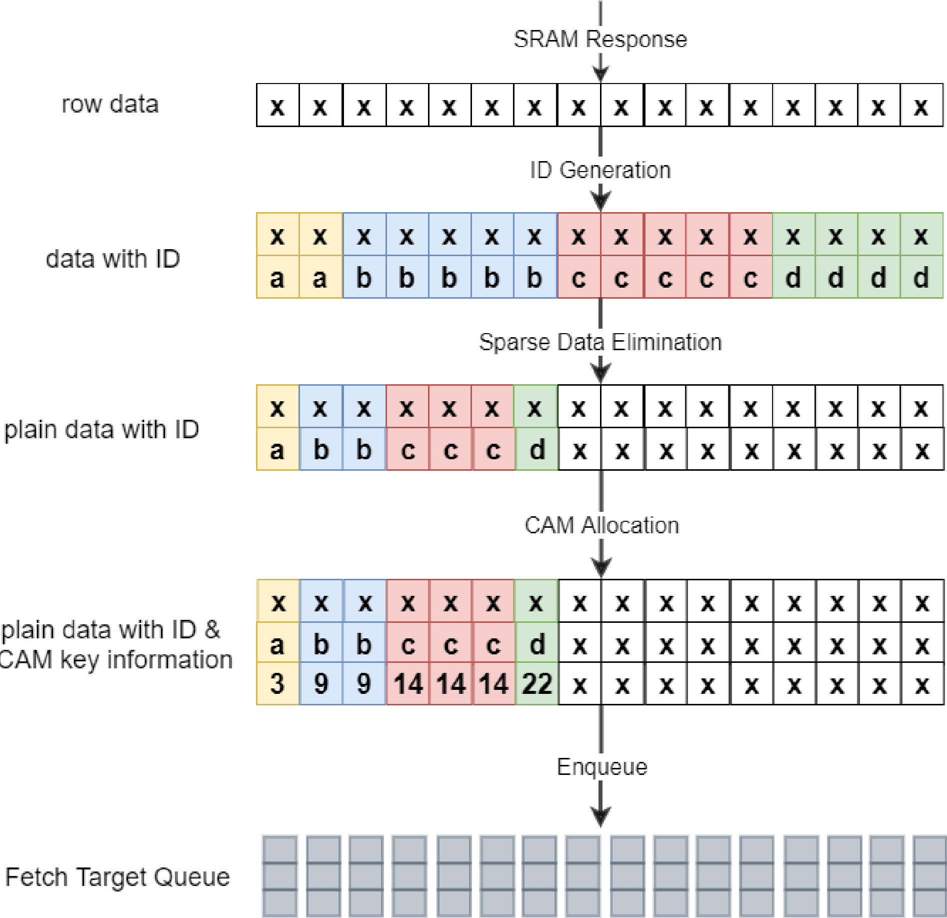


Figure 5.2: An example fetch unit execution.

An example fetch unit execution is given in Figure 5.2. First, identity assignment for a neuron vector size of five is performed by an identity generator. Then, the sparsity controller eliminates zero values from the vector. The resulting CAM key information together with identity and neuron vector is written to the fetch target queue. When fetch target queue is full, fetch is stalled until stall condition is resolved.

5.2 Neuron Vector Table (NVT)

This module is composed of a content addressable memory (CAM) and two auxiliary components for allocation and deallocation, as shown in Figure 5.3. Each row in CAM represents a neuron vector.

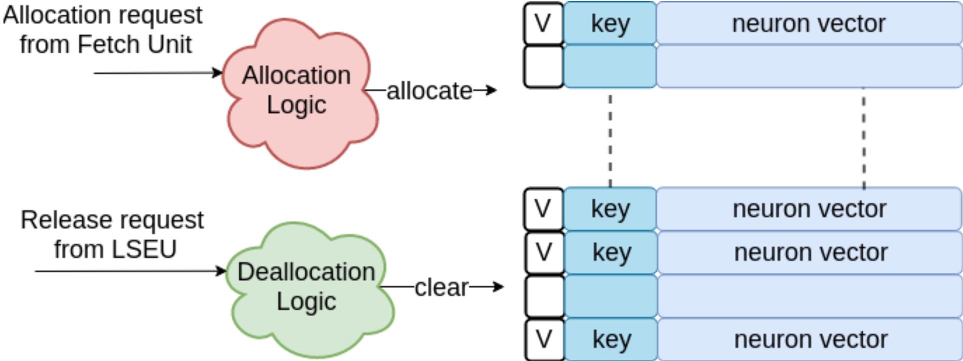


Figure 5.3: Banked content addressable memory and neuron vector allocation.

When a new identity is generated by the fetch module, a key is allocated and the neuron vector is stored at that location. In order to process different neuron vectors in the same cycle, allocation logic has the ability to allocate multiple keys. In addition, depending on the vector dimension for a given network and fetch size, all neuron vectors might not be available in a single cycle. In such cases, this logic is able to append the remaining data of a neuron vector to the previously allocated key.

When the load-store-execute unit requests a neuron vector from the table, data corresponding to the key is sent and the whole entry is deleted by deallocation logic. The banked memory system in the load-store-execute unit imposes multiple key release requirements on deallocation logic in order to perform parallel processing and storage. For M banks, our deallocation logic is capable of freeing M keys during the same cycle.

5.3 Reduction Unit (RU)

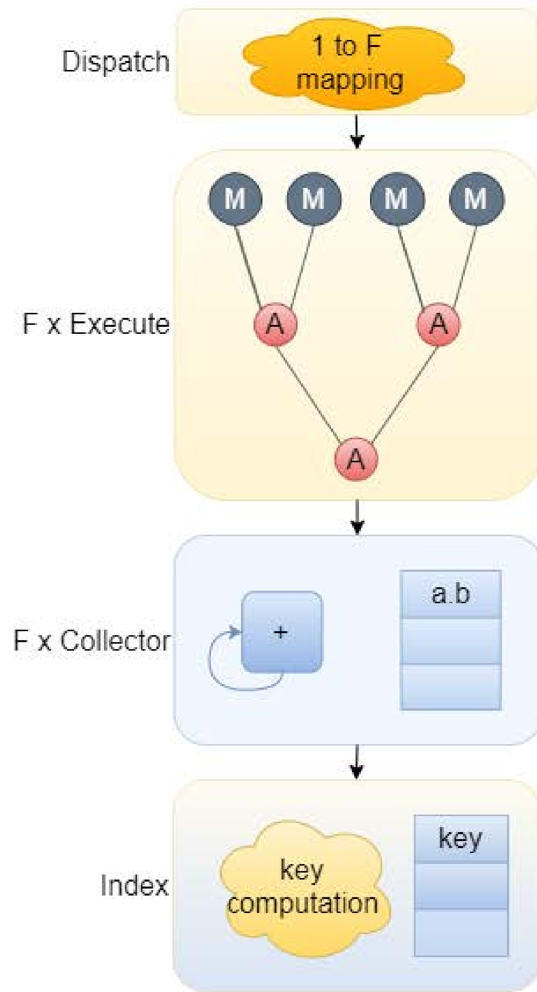
The reduction unit performs dot product between neuron vectors and previously-stored hash tables. It is composed of four modules, namely, dispatch, execute, collector, and index, as shown in Figure 5.4.

In the first step, the dispatch unit sends neuron vectors retrieved from the fetch target queue to execution units according to the given hash size setting. When there is no available execute unit, fetch dequeue is put on hold.

The execution unit is at the heart of this accelerator design as it significantly determines performance and runtime. For high performance, it is most critical to perform multiple dot product operations in the same cycle with using resources efficiently. Among many reduction trees, SIGMA's forwarding adder network (FAN) [23] is used because of its ability to process different identities in the same cycle with the highest throughput possible. In this micro-architecture, shown in Figure 5.5, there exists a tree-based reduction network. It provides parallel processing by means of forwarding adders. At each stage, inputs of an adder are determined by two multiplexers. Control bits of these multiplexers are selected by identities of neuron vectors. The algorithm used to determine control bits of multiplexers is given in [23].

Because of the irregular data fetch and various vector size settings, a neuron vector may need to be split and processed in different cycles. Collector, next sub-module, checks if the dot product is completely finished for each identity. When the operation is in progress, results from the previous cycle are cumulatively stored in registers inside this module.

After the collector completes its task, the dot product result is sent to the index module. This unit implements hashing function described in Section 4.1. In other words, the cluster key is computed by this module. Then, the cluster key along with the identity and CAM key information is stored in a queue.



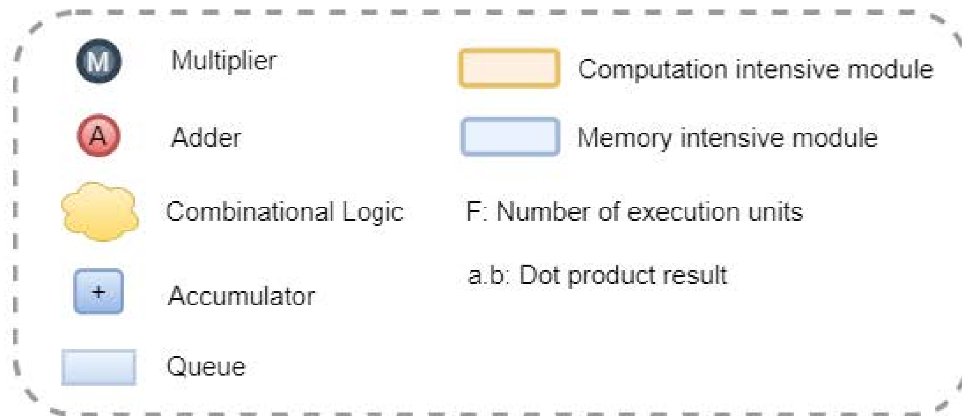


Figure 5.4: Details of the reduction unit with four modules, namely dispatch, execute, collector, and index.

Internal architecture for a forwarding adder network with sixteen multipliers and fifteen adders and an example execution scenario for the whole stage is given in Figure 5.5. First, fetched neuron vectors with identity and destination information are dispatched into forwarding adder networks according to the hash size setting. Then, multiplication operation is performed between hash tables and neuron networks. After multiplication, depending on the identity, data is forwarded to adders across stages to fulfill correct dot product operation. When a neuron vector is reached to its predetermined destination, results are accumulated for a neuron vector by the collector sub-module. Finally, the index module generates the cluster key using dot product results and stores it in a queue.

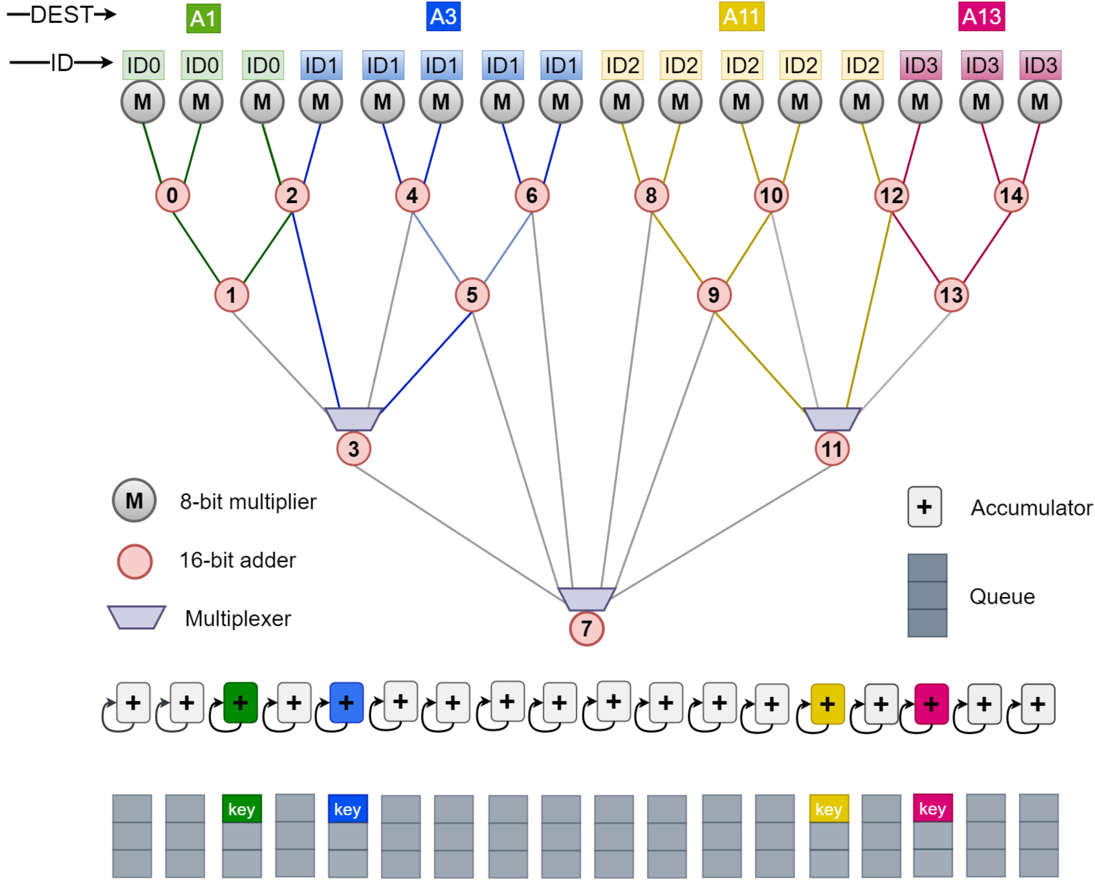


Figure 5.5: Forwarding adder network topology with collector and index modules.

5.4 Dispatch Unit (DU)

The dispatch unit includes two auxiliary sub-modules as bank assignment unit and compactor, which will be explained in the following paragraphs.

As shown in Figure 5.6, at the end of the reduction unit, cluster keys are generated and written to the collector queues. This module gets all generated keys from the queues and assigns a bank for each valid key according to a specific algorithm. For example, the algorithm can select an available bank according to the most significant bits of the keys. Here, it is important to note that the algorithm should produce a uniform distribution of bank numbers to maximize parallel memory access and minimize stalls along the pipeline. In our design, for M banks, the bank assignment sub-module performs selection according to the least significant log_2M bits of the cluster keys. This way we could get a uniform distribution of banks.

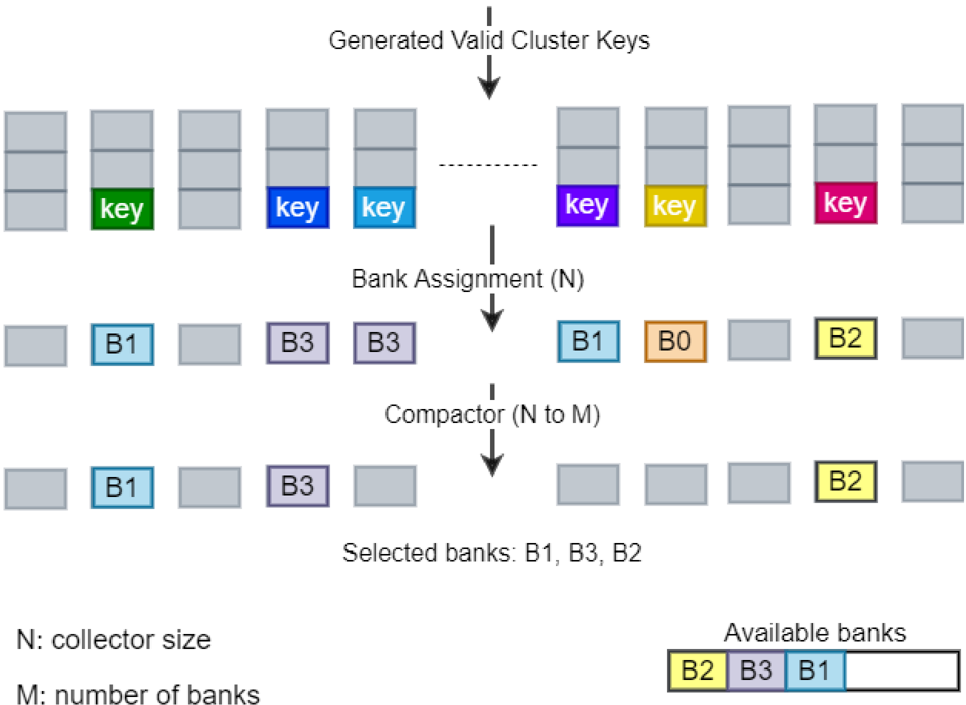


Figure 5.6: Example execution scenario for the dispatch unit with N collectors and M banks.

In some cases, it is possible to have more keys fetched from the reduction unit than available banks in the load-store-execute unit. Besides, it is also possible that more than one key could be assigned to the same bank. Therefore, it is necessary to perform selection among such conflicts. In our design, the compactor sub-module connects the first k of n valid cluster keys to k available banks, thereby, avoiding possible conflicts.

5.5 Load-Store-Execute Unit (LSEU)

This unit is designed to perform centroid calculation and storage. It contains four submodules, namely, bank request buffer, memory units, centroid execution unit, and controller, as shown in Figure 5.7.

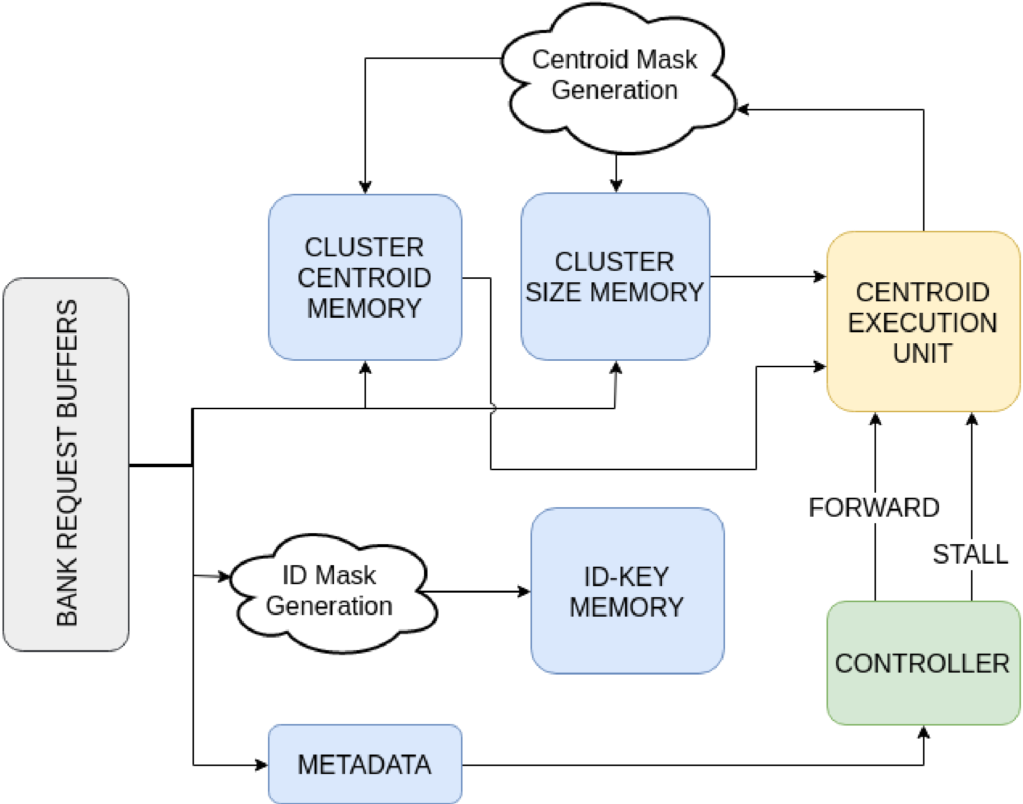


Figure 5.7: The load-store-execute unit (LSEU) is composed of cluster memories, ID-Key memory, and centroid execution unit

During the first stage of this module, dispatched keys along with identity information are enqueued to their predetermined bank request buffers. At the same time, the neuron vector is read from the table using the associated key.

In the next step, previous centroid information and cluster size are read from the cluster centroid memory and the cluster size memory, respectively. In addition, metadata is read to check if cluster has been formed before. These three memories are necessary to update the cluster centroid correctly. Besides, cluster key information is written into identity memory for later use by the full result derivation stage mentioned in Chapter 3.

In the third stage, the centroid execution unit updates cluster centroid using data from previous cycles and metadata. This operation has to be performed atomically. Specifically, read, modify, and write operations must be in the same cycle. However, due to limitations in memory technology, each read operation takes at least one cycle. As such, we guarantee atomicity by forwarding updated centroids across pipeline stages, thereby, maintaining functional correctness.

In order to use memory efficiently, we use customized SRAMs for cluster centroid, cluster size, and identity-key memories. Specifically, they have the ability to mask data to be written in byte granularity. In addition, special mask generation circuitry for read and write operations in front of the SRAMs helps to provide compatibility with any kind of convolutional neural networks through reconfiguration. Furthermore, banked access to all SRAMs allows parallel execution. All these features are critical in order to get high performance, energy efficiency, and resource utilization.

5.6 Controller

The controller unit enables the host to modify the network-related parameters such as hash size, vector dimension, and start address of SRAM for each layer of a network. All modules must work in coordination with the controller to achieve dynamic reconfiguration. For example, fetch unit, reduction unit, load-store-execute unit, and CAM have the capability of handling the neuron vectors for different vector and hash sizes. For compatibility with any kind of convolutional neural network, it is critical to achieve this online reconfiguration. Besides compatibility, online reconfiguration also provides resource efficiency and speedup by parallel processing of neuron vectors.

5.7 Discussion

In this section, we give a discussion of ways to further improving the architecture to get additional performance and energy efficiency. For this purpose, we propose and explain two methods: 1) folding support in the fetch unit and 2) unfolding engine.

5.7.1 Folding Support in Fetch Unit

After zero removal inside the fetch unit by the sparsity controller, non-zero data may leave more than half of the rows empty. By means of a new submodule to be added, it can be first checked whether the processed non-zero vector received from the sparsity controller is foldable. If possible, it is folded and stored in a queue. Folding provides efficient use of resources, thereby increasing performance.

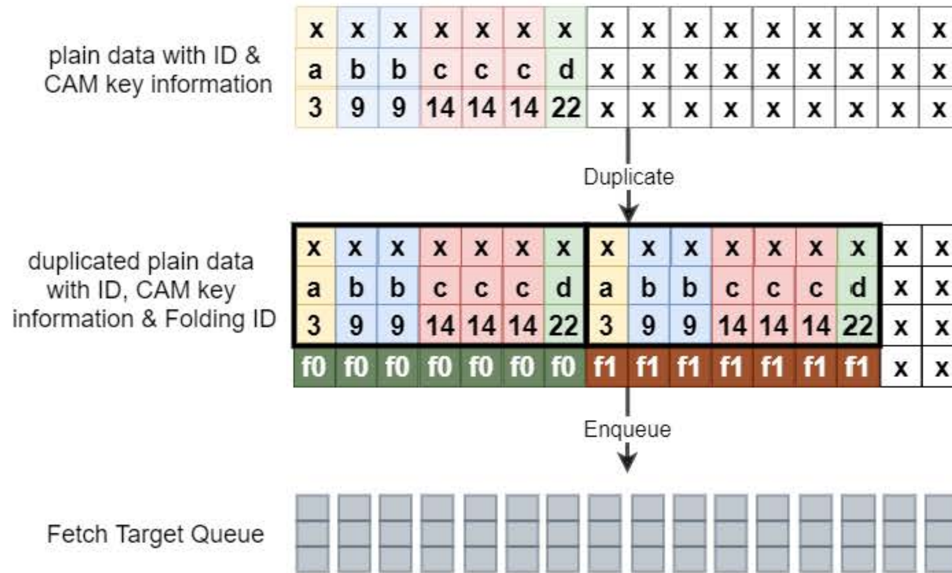


Figure 5.8: Fetch unit supporting fold operation.

As can be seen in the example given in Figure 5.8, the fetched row has seven non-zero and nine zero values. Thus, it can be duplicated and written to the fetch target queue together with the folding information. This information is used by the reduction unit and enables the reduction unit to generate cluster keys with half the number of execution sub-modules. However, it is important to note that the current reduction tree [23] is not able to handle duplicate identities. Therefore, it has to be modified respectively.

5.7.2 Unfolding Engine

As described in Chapter 3, the host performs an unfolding operation to implement convolution as a matrix multiplication. Since this operation requires negligible memory and computation, it can be easily integrated into the pipeline. This way it is possible to access the input more efficiently, saving energy and increasing performance.

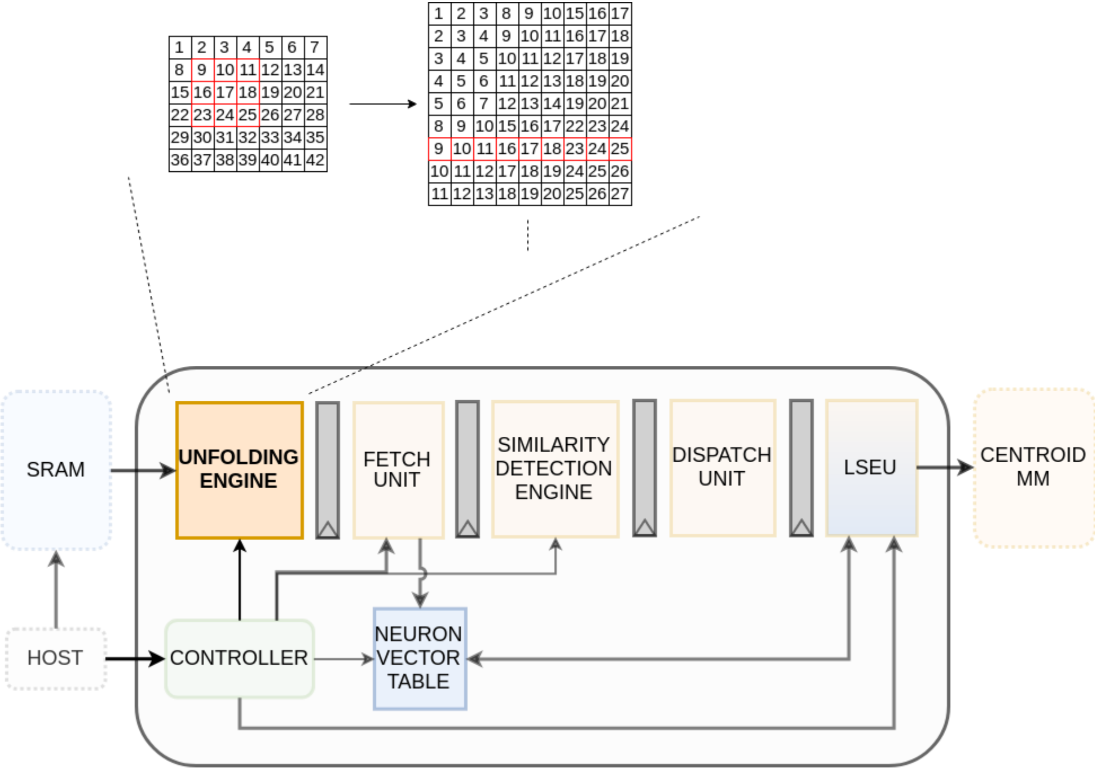


Figure 5.9: Advanced architecture supporting on the fly unfolding operation.

The architectural block diagram and an example unfolding operation are given in Figure 5.9. As can be seen from this figure, the unfolding module sits in between the SRAM and fetch unit. In this figure, the highlighted region on the matrix represents a convolution with the unfolded vector on the right. The reused data on the right-hand side shows the potential in terms of efficiency.

5.8 Synergy with Other CNN Accelerators

This section explains how the *general reuse discovery engine* can be integrated into an existing hardware accelerator. An example integration into Eyeriss [2] is given in Figure 5.10.

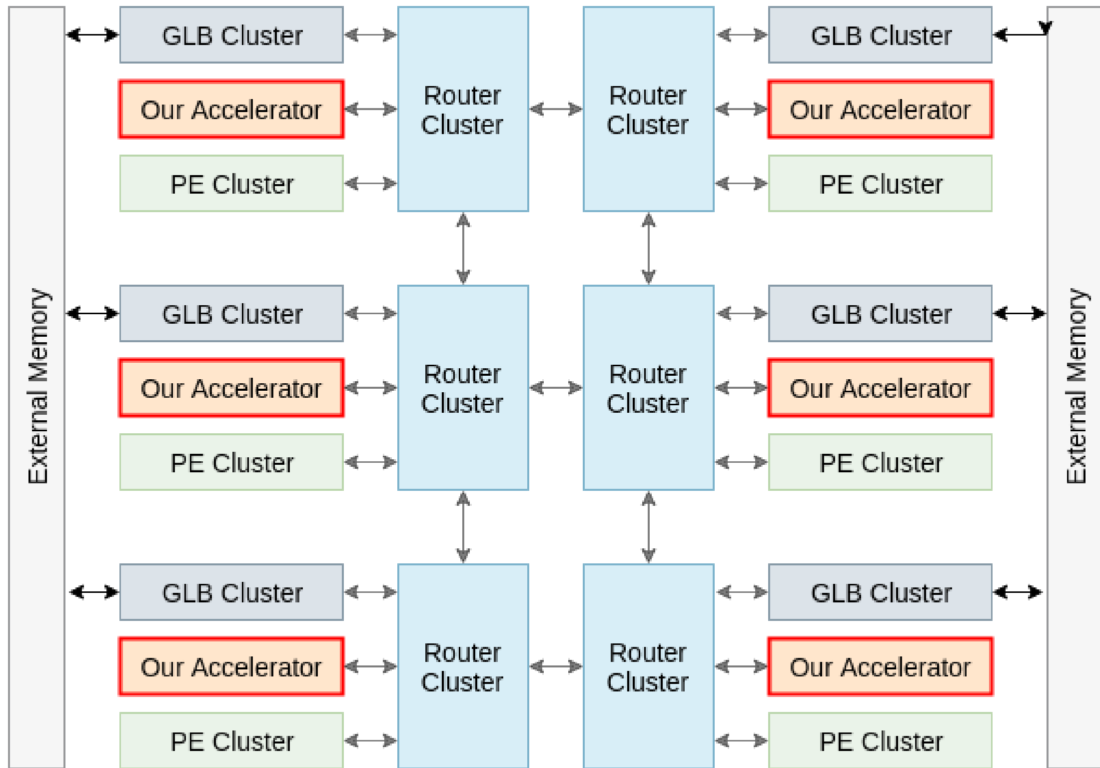


Figure 5.10: Top level architecture for the integration of the reuse-centric accelerator into Eyeriss [2].

Hardware accelerators generally contain a global buffer to keep input activations, and results [2, 24, 17, 23]. Our accelerator can be placed in front of this global buffer to act as a preprocessor. This way, the size of the input layer can be reduced by extracting similarities. Then, a reduced centroid matrix is fed into processing engines to calculate the layer output. Finally, the full result is derived by reading local ID-Key memory inside our accelerator, and it is stored in the global buffer for processing in the next layer.

Eyeriss architecture [2] has a 2D mesh network consisting of global clusters, PE clusters, and router clusters. In this Eyeriss integration, our accelerator is connected to *router cluster* in order to fetch raw layer inputs from *global cluster* and send centroid matrix to *PE cluster* as shown in Figure 5.10. It is important to note that the scalability of the design enables us to fit the accelerator into any existing framework.

Chapter 6

Evaluation

To evaluate the proposed accelerator's efficiency, we test the benefits that it may bring to it with both software-based CNN accelerations and other CNN hardware accelerators. Energy efficiency and the speedup are used as a metric to represent an improvement.

6.1 Experimental Setup

This section describes the tools, benchmarks, data sets, default parameters, and execution environments used to evaluate our accelerator.

6.1.1 Tools

We design the accelerator described in Chapter 5 using Chisel[25] hardware construction language. Then, we measure performance by running generated RTL from Chisel on a cycle-accurate simulator called Verilator [26]. After that, we synthesize and implement our accelerator using Xilinx's Vivado [27] tool for VCU118 [28] evaluation board to obtain the average energy consumption and

resource utilization. Furthermore, to show potential benefits from integration, we use Scale-Sim [29, 30] tool. We measure the speedup when our accelerator is integrated into Eyeriss [24].

6.1.2 Benchmarks

We use four convolutional neural networks to evaluate our accelerator, namely, CifarNet [31], AlexNet [32], VGG-19 [33], and MobileNet [34]. Table 6.1 lists datasets running on described networks.

Table 6.1: Convolutional network and dataset pairs used in evaluation.

Network	**Dataset**
CifarNet	Cifar10
AlexNet	Imagenet
VGG-19	Imagenet
MobileNet	Imagenet

6.1.3 Data Sets

We use Cifar10 [31] and Imagenet [35] as our datasets for collecting the experimental results. Cifar10 has 60000 images with a size of 32x32, whereas the Imagenet dataset has more than 14 million images with a size of 224x224.

6.1.4 Default Parameters

The default architecture related parameters for our advanced accelerator are given in Table 6.2. These optimal architectural parameters are selected for the default setup considering performance, energy, and area.

The default CNN layer related parameters for each benchmark is given in Table 6.3. Note that, it is necessary to adjust the hash size and vector size online for each layer without compromising accuracy.

Table 6.2: Default parameters used for the advanced architecture.

Parameter	Value
Data precision	8-bits
SRAM row width	128-bits
Fetch target queue depth	8
FAN size	16
Number of FANs	16
FAN/Bank request queue depth	8
Cluster memory size	24 MB
ID memory size	2.5 MB
Count memory size	2.5 MB
Valid memory size	1 MB
Maximum hash size	20
Maximum vector dimension	24
Number of Banks	4
CAM lines	64

6.1.5 Execution Environments

We tested the reuse-based implementation in four different execution environments as listed below.

1. **CPU:** Software-based implementation running on a mobile CPU. This is used as a baseline for our experimental evaluations.
2. **Basic Accelerator:** Our basic reuse-centric architecture as explained in Chapter 4.
3. **Advanced Accelerator:** Our advanced accelerator architecture as explained in Chapter 5.
4. **Eyeriss + Advanced Accelerator:** This is the setup where our accelerator is integrated into an existing state-of-the-art accelerator, Eyeriss [24], in order to observe potential benefits in runtime. We use SCALE-sim [29, 30], a CNN accelerator simulator, to measure the performance.

Table 6.3: Algorithm specific parameters used for the benchmarks.

Network	**Layer Number**	**Batch Size**	**Input Vectors**	**Vector Size**	**Hash Size**	**Sub-vector**
CIFARNET	conv1	100	1024	5	15	15
CIFARNET	conv2	100	256	10	10	160
ALEXNET	conv1	100	2916	11	16	33
ALEXNET	conv2	100	676	20	15	80
ALEXNET	conv3	100	144	12	15	144
ALEXNET	conv4	100	144	12	15	288
ALEXNET	conv5	100	144	24	15	144
VGG-19	conv1	16	50176	9	20	3
VGG-19	conv2	16	50176	16	20	36
VGG-19	conv3	16	12544	16	18	36
VGG-19	conv4	16	12544	16	18	72
VGG-19	conv5	16	3136	16	16	72
VGG-19	conv6	16	3136	16	16	144
VGG-19	conv7	16	3136	16	16	144
VGG-19	conv8	16	3136	16	16	144
VGG-19	conv9	16	784	16	15	144
VGG-19	conv10	16	784	18	15	256
VGG-19	conv11	16	784	18	15	256
VGG-19	conv12	16	784	18	15	256
VGG-19	conv13	16	196	18	12	256
VGG-19	conv14	16	196	18	12	256
VGG-19	conv15	16	196	18	12	256
VGG-19	conv16	16	196	18	12	256
MOBILENET	conv1	4	12544	18	3	9
MOBILENET	conv3	4	12544	18	4	8
MOBILENET	conv5	4	3136	18	4	16
MOBILENET	conv7	4	3136	18	8	16
MOBILENET	conv9	4	784	10	8	16
MOBILENET	conv11	4	784	10	8	32
MOBILENET	conv13	4	196	10	8	32
MOBILENET	conv15	4	196	10	8	64
MOBILENET	conv17	4	196	10	8	64
MOBILENET	conv19	4	196	10	8	64
MOBILENET	conv21	4	196	8	8	64
MOBILENET	conv23	4	196	8	8	64
MOBILENET	conv25	4	49	8	8	64
MOBILENET	conv27	4	49	8	16	64

6.2 Performance and Synergy with Existing Accelerators

Previous studies [13] have shown that the effect of reuse-centric CNN acceleration on the original accuracy of the network is minimal, generally much less than 1%. Therefore, this section will focus on performance evaluation for all execution environments.

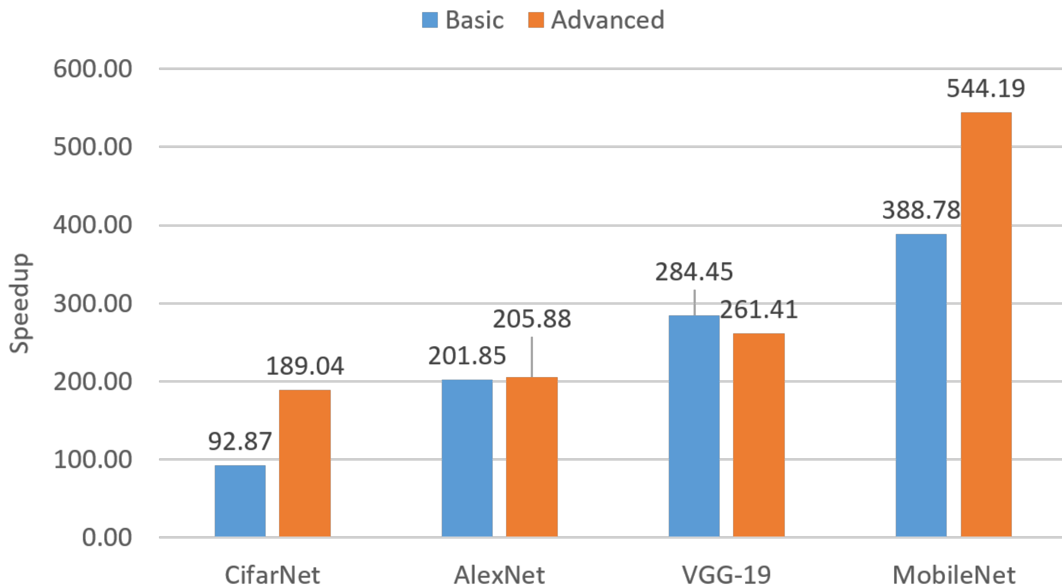


Figure 6.1: Average performance improvement over mobile CPU obtained through basic and advanced design for the similarity discovery module.

The first set of results highlight that our proposed basic and advanced accelerator implementations offer great speedups, as shown in Figure 6.1. For CifarNet, advanced design performs almost two times better than basic design while they are close to one another for AlexNet and VGG-19. The reason behind this is related to the setting for the number of FANs parameter described in Table 6.2. As shown in Table 6.3, VGG-19 has hash sizes greater than 16 for the first four layers. Thus, it can process a single neuron vector in two cycles for the current setting, resulting in similar performance results for the advanced and basic designs. Similar to VGG-19, AlexNet has vector sizes between 10 and 24 as given in Table 6.3, which means that fetching a single neuron vector requires at least one

cycle. On the other hand, for CifarNet and MobileNet, our advanced accelerator can process more than one neuron vector in the same cycle, resulting in much larger speedups than basic design.

Second, we analyzed speedups of convolutional layers for the CPU+accelerator platform as reported in Figure 6.2. In this setup, similarity detection is performed on our advanced accelerator, while centroid multiplication and full result derivation are performed on the CPU. As can be observed from the figure, the performance improvement is ranging between 1.46X and 3.63X. However, full result derivation and centroid multiplication still create a bottleneck.

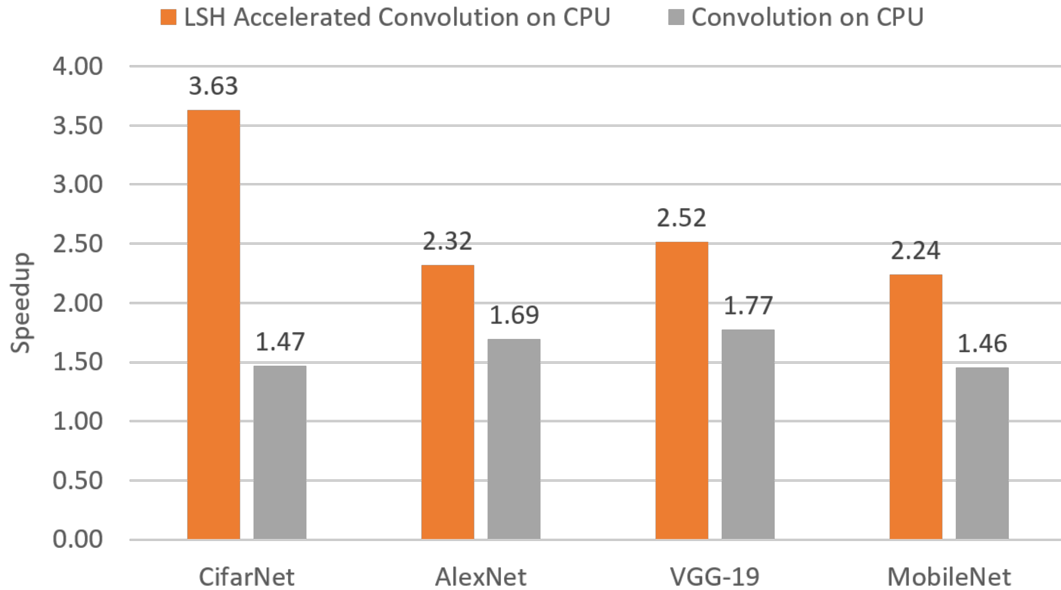


Figure 6.2: Performance improvement of reuse-centric implementations on CPU and with the proposed accelerator. The baseline is the default convolution performed on a mobile CPU.

We further integrate our accelerator into Eyeriss to illustrate the benefits brought by our accelerator to the existing state-of-the-art hardware accelerator. We use SCALE-sim to simulate the performance of Eyeriss, and the execution cycles of running the entire convolutional layer on Eyeriss is used as the baseline. As an integrated unit, we implement the *reuse centric* CNN acceleration with a combination of Eyeriss and our accelerator. The similarity discovery module

is implemented on our accelerator while Eyeriss processes the centroid matrix-multiplication and full-result derivation. Figure 6.3 illustrates the performance comparison between the baseline and the integrated unit. The results show that our proposed accelerator provides up to 7.69X speedups and very promising average speedups for all of the benchmarks, largely boosting the performance.

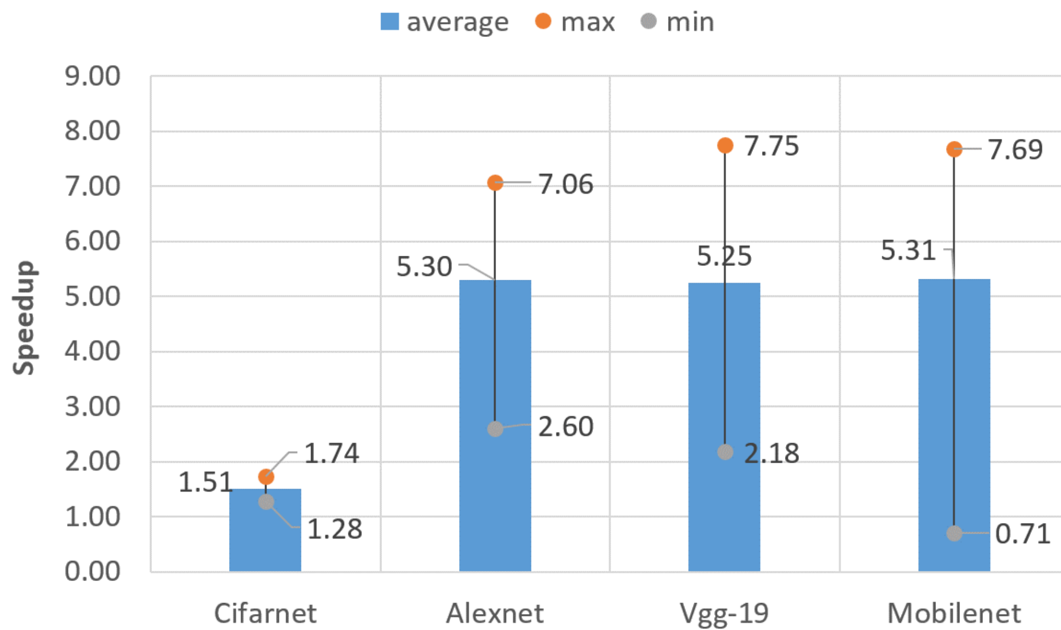


Figure 6.3: The performance improvement brought to Eyeriss by our reuse-centric accelerator.

6.3 Energy Reduction

We perform synthesis using Xilinx's Vivado tool [27] to measure the power consumption of the advanced design on VCU118 [28] FPGA development board. We compare our accelerator's energy consumption with the software-based implementation running on CPU as shown in Figure 6.4. Specifically, this figure presents the energy efficiency over mobile CPU for the similarity discovery module. As can be seen from these results, our accelerator reduces the energy consumption up to 95.46%.

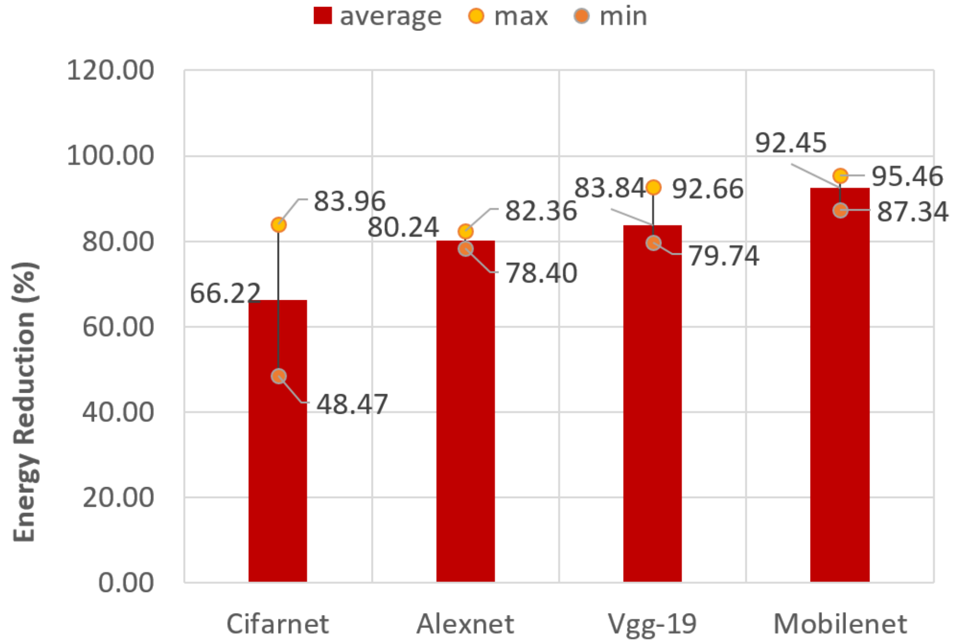


Figure 6.4: Energy reduction over mobile CPU through advanced design.

6.4 Area Breakdown and Resource Utilization

Cluster memory size is too large to be implemented in an FPGA synthesis for the given parameters in Table 6.2. Therefore, we choose smaller sizes for SRAMs and perform technological mapping of our design to FPGA's logic resources through

synthesis. After synthesis, the area breakdown of the modules in advanced architecture based on the lookup tables (LUTs) and registers is given in Figure 6.5. As can be seen from this figure, the resources are largely occupied by the reduction unit's execution module, which consists of a set of forwarding adder networks (FANs). We give specific details about the resources used by each module in Table 6.4.

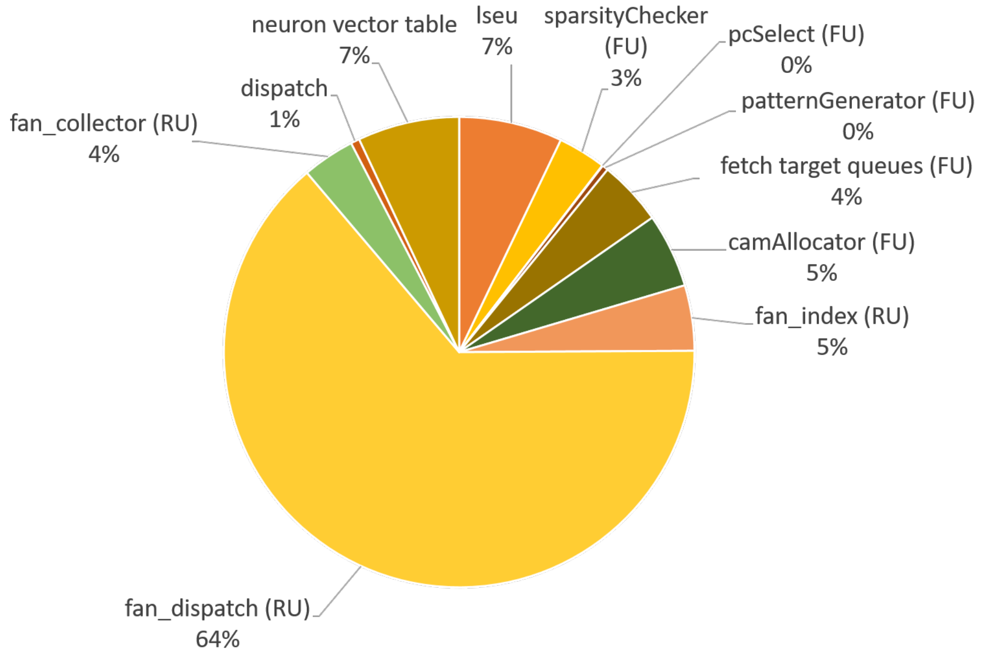


Figure 6.5: Area breakdown of the modules in the advanced architecture based on the lookup tables (LUTs) and registers.

Table 6.4: Resources used by each module of the advanced design implemented on VCU118 board.

Module	CLB LUTs	CLB Registers	Carry8	F7 MUXs	Block RAM Tile
LSEU	15,064	1,752	1,352	0	68
FU	31,277	446	1,807	20	0
RU	84,038	86,255	2,080	1	0
DU	1,425	0	0	0	0
NVT	14,976	1,551	0	1,536	0

6.5 Discussion

In the load-store-execute unit, updated centroids are written to the cluster centroid memory based on the generated key information. The number of unique keys determines the number of centroids, namely the centroid memory's actual size. On the other hand, we set a theoretical size for centroid memory to include all possible keys. When the number of centroids for the aforementioned benchmarks given in Table 6.5 are considered, one can observe that there is a huge difference between the theoretical size and the actual size of the cluster centroid memory. With this observation, we opt to use a much smaller memory together with a special hashing function to minimize collisions.

Table 6.5: Comparison of theoretical size with actual size for cluster memory.

Network	**Layer**	**Theoretical Size (MB)**	**Actual Size (KB)**	**Difference**
CIFARNET	conv1	0.16384	6.144	27.31
CIFARNET	conv2	0.01024	0.3072	34.13
ALEXNET	conv1	0.720896	96.228	7.67
ALEXNET	conv2	0.65536	47.32	14.18
ALEXNET	conv3	0.393216	12.2688	32.82
ALEXNET	conv4	0.393216	12.096	33.29
ALEXNET	conv5	0.786432	59.0976	13.63
VGG-19	conv1	9.437184	187.858944	51.44
VGG-19	conv2	16.777216	206.8054016	83.07
VGG-19	conv3	4.194304	79.9604736	53.71
VGG-19	conv4	4.194304	104.6872064	41.03
VGG-19	conv5	1.048576	59.9703552	17.90
VGG-19	conv6	1.048576	51.2999424	20.93
VGG-19	conv7	1.048576	47.2055808	22.75
VGG-19	conv8	1.048576	33.4774272	32.07
VGG-19	conv9	0.524288	20.672512	25.97
VGG-19	conv10	0.589824	27.320832	22.11
VGG-19	conv11	0.589824	28.449792	21.23
VGG-19	conv12	0.589824	30.256128	19.96
VGG-19	conv13	0.073728	10.44288	7.23
VGG-19	conv14	0.073728	9.03168	8.36
VGG-19	conv15	0.073728	7.733376	9.76
VGG-19	conv16	0.073728	7.056	10.70
MOBILENET	conv1	0.786432	0.75264	1069.98
MOBILENET	conv3	1.048576	0.0802816	13374.69
MOBILENET	conv5	1.048576	0.2709504	3962.87
MOBILENET	conv7	2.097152	4.6362624	463.19
MOBILENET	conv9	0.008192	0.4164608	20.14
MOBILENET	conv11	0.008192	0.777728	10.79
MOBILENET	conv13	0.008192	0.392	21.40
MOBILENET	conv15	0.008192	0.5375104	15.61
MOBILENET	conv17	0.008192	0.4785536	17.53
MOBILENET	conv19	0.008192	0.410816	20.42
MOBILENET	conv21	0.002048	0.2546432	8.24
MOBILENET	conv23	0.002048	0.254016	8.26
MOBILENET	conv25	0.002048	0.105056	19.96
MOBILENET	conv27	0.004096	0.404544	10.37

Chapter 7

Related Work

Studies seeking opportunities to maximize performance and to minimize energy consumption can be grouped into two main categories as improvements in hardware architectures and joint hardware/software designs.

From the architectural perspective, there are four kinds of architecture exploiting different data reuse characteristics: no local reuse, weight stationary, output stationary, and row stationary [19]. First, no local reuse (NLR) type architectures do not store input, output, and filter data locally inside a processing engine; instead, they use large global buffers to handle dataflow. An example architecture in this approach is DaDianNao in [36], which uses a large on-chip memory (eDRAM) to buffer input, output, and weights instead of off-chip memory access which is highly costly. In weight stationary dataflow, filter weights are generally stored near processing engines to decrease their access cost. Input feature maps are mapped to all PEs such that stored weights' utilization is maximized. An example of this architecture is nn-X [37], which caches the incoming weights to use during the convolution operation. Similarly, filter weights of CNN accelerators in [38] and [39] are kept in registers of their PEs to perform the convolution. Partial sums can also be stored inside local register files, resulting in another architecture type called output stationary. According to the mapping to PEs, there

can be different types of dataflows targeting a different number of output channels and activations. For instance, in ShiDianNao [40], computation is brought near to the sensor to eliminate the large bandwidth requirement due to DRAM and each PE provides an output activation by storing partial sums in a register, targeting calculations of all output activations for a single channel. The latest approach, proposed in [19] called row stationary (RS), suggests maximum reuse of all types of data, namely input feature maps, filter weights, and partial sums inside a processing engine. Eyeriss [24] is a DNN accelerator that is implemented in ASIC and exploits this approach. In this architecture, an input row, a filter row, and partial sum stay stationary inside local register files so that 1D convolution can be performed in a single PE. Our accelerator provides additional benefits for different kind of data reuse characteristics when integrated into an existing accelerator.

From the hardware/software co-design perspective, one can take advantage of algorithmic properties to design simpler hardware besides improving performance and energy efficiency. This can be primarily done by reducing precision or reducing computation size without sacrificing accuracy [41]. Methods for reducing precision can be implemented by quantizing data either uniformly or non-uniformly, i.e., it is possible to use equal or different intervals between the quantization levels. Firstly, uniform quantization can be used to convert 32-bit floating-point data to an 8-bit dynamically fixed point [42] while maintaining accuracy. An example accelerator using 8-bit integer arithmetic is Google's Tensor Processing Unit [43]. Precision can be reduced to only one or two bits as implemented in YodaNN [44] and in XNOR-Net [45]. These hardware accelerators, also called binary networks, offer extremely low-cost hardware along with some accuracy degradation. As for the reduced computation size, the sparsity, resulting from ReLU result in output feature map, can be used for saving energy and reducing implementation cost of a DNN hardware. Using compression on the sparse matrix and skipping zero-valued features in calculations can provide great energy efficiency [24]. Besides our accelerator working with 8-bit data and supporting sparsity check, determining clustering parameters with no loss in accuracy and for efficient inference sets a good example of hardware/software co-design.

There are also recent works on hardware accelerators leveraging computation reuse. For example, Riera et al. [10] investigate similarities between consecutive frames. Specifically, they miss the similarities across frames at different layers or batches. The study by Buckler et. al. [46] proposes an algorithm for efficient processing of real time vision. This algorithm estimates motion in the input frame to predict the next frame. Another work by Hedge et al. [11] improves performance and saves energy by reusing weights in and across filters. They propose a computation reuse scheme based on weight repetition and reducing CNN model size. As a result, [10] and [46] only makes use of temporal reuse opportunities while [11] suggests spatial reuse to reduce computation and memory reads. On the other hand, our general reuse-centric CNN accelerator take advantage of both temporal and spatial reuse for better performance and energy.

Chapter 8

Conclusion

In this thesis, we propose an architecture that reuses similar neuron vectors in order to boost the performance of CNN executions. Our accelerator design is flexible, scalable, and resource-efficient to enable integration into any existing accelerator and to speedup CNN inference. We measured its performance through four CNNs and integrated it into Eyeriss [24]. Specifically, it provides speedups up to 7.75X for a convolutional layer. Furthermore, we also integrated into a software-based implementation of general reuse-centric CNN acceleration. We observe up to 3.63X faster execution in a convolutional layer and we save energy up to 95.46% in similarity detection for the same CNN.

Bibliography

[1] L. Ning, *Deep Reuse for Deep Learning*. PhD thesis, North Carolina State University, 2020.

[2] Y. Chen, J. S. Emer, and V. Sze, "Eyeriss v2: A flexible and high-performance accelerator for emerging deep neural networks," *CoRR*, vol. abs/1807.07928, 2018.

[3] A. Lavin and S. Gray, "Fast algorithms for convolutional neural networks," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 4013–4021, 2016.

[4] M. Mathieu, M. Henaff, and Y. LeCun, "Fast training of convolutional networks through ffts: International conference on learning representations (iclr2014), cbls, april 2014," Jan. 2014. 2nd International Conference on Learning Representations, ICLR 2014 ; Conference date: 14-04-2014 Through 16-04-2014.

[5] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding," in *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings* (Y. Bengio and Y. LeCun, eds.), 2016.

[6] J. Yu, A. Lukefahr, D. Palframan, G. Dasika, R. Das, and S. Mahlke, "Scalpel: Customizing dnn pruning to the underlying hardware parallelism," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, 2017.

[7] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "Eie: Efficient inference engine on compressed deep neural network," in *Proceedings of the 43rd International Symposium on Computer Architecture*, ISCA '16, pp. 243–254, 2016.

[8] M. Gao, J. Pu, X. Yang, M. Horowitz, and C. Kozyrakis, "Tetris: Scalable and efficient neural network acceleration with 3d memory," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, (New York, NY, USA), p. 751–764, Association for Computing Machinery, 2017.

[9] H. Sharma, J. Park, D. Mahajan, E. Amaro, J. K. Kim, C. Shao, A. Mishra, and H. Esmaeilzadeh, "From high-level deep neural models to fpgas," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016.

[10] M. Riera, J.-M. Arnau, and A. González, "Computation reuse in dnns by exploiting input similarity," in *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ISCA '18, (Piscataway, NJ, USA), pp. 57–68, IEEE Press, 2018.

[11] K. Hegde, J. Yu, R. Agrawal, M. Yan, M. Pellauer, and C. W. Fletcher, "Ucnn: Exploiting computational reuse in deep neural networks via weight repetition," in *Proceedings of the 45th Annual International Symposium on Computer Architecture*, 2018.

[12] Y. Shen, M. Ferdman, and P. Milder, "Maximizing cnn accelerator efficiency through resource partitioning," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, 2017.

[13] L. Ning and X. Shen, "Deep reuse: Streamline cnn inference on the fly via coarse-grained computation reuse," in *Proceedings of the ACM International Conference on Supercomputing*, ICS '19, 2019.

[14] L. Ning, H. Guan, and X. Shen, "Adaptive deep reuse: Accelerating cnn training on the fly," in *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, 2019.

[15] A. Andoni, P. Indyk, T. Laarhoven, I. Razenshteyn, and L. Schmidt, "Practical and optimal lsh for angular distance," in *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 1*, (Cambridge, MA, USA), pp. 1225–1233, MIT Press, 2015.

[16] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni, "Locality-sensitive hashing scheme based on p-stable distributions," in *Proceedings of the Twentieth Annual Symposium on Computational Geometry*, (New York, NY, USA), pp. 253–262, ACM, 2004.

[17] V. Sze, Y. Chen, T. Yang, and J. S. Emer, "Efficient processing of deep neural networks: A tutorial and survey," *Proceedings of the IEEE*, vol. 105, pp. 2295–2329, Dec 2017.

[18] J. Cong and B. Xiao, "Minimizing computation in convolutional neural networks," in *Artificial Neural Networks and Machine Learning – ICANN 2014* (S. Wermter, C. Weber, W. Duch, T. Honkela, P. Koprinkova-Hristova, S. Magg, G. Palm, and A. E. P. Villa, eds.), (Cham), pp. 281–290, Springer International Publishing, 2014.

[19] Y. Chen, J. Emer, and V. Sze, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pp. 367–379, 2016.

[20] P. Indyk and R. Motwani, "Approximate nearest neighbors: towards removing the curse of dimensionality," in *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, pp. 604–613, 1998.

[21] A. Andoni and P. Indyk, "Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions," in *Foundations of Computer Science, 2006. FOCS'06. 47th Annual IEEE Symposium on*, pp. 459–468, 2006.

[22] K. Terasawa and Y. Tanaka, "Spherical lsh for approximate nearest neighbor search on unit hypersphere," in *Workshop on Algorithms and Data Structures*, pp. 27–38, 2007.

[23] E. Qin, A. Samajdar, H. Kwon, V. Nadella, S. Srinivasan, D. Das, B. Kaul, and T. Krishna, "Sigma: A sparse and irregular gemm accelerator with flexible interconnects for dnn training," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 58–70, 2020.

[24] Chen, Yu-Hsin and Krishna, Tushar and Emer, Joel and Sze, Vivienne, "Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks," in *IEEE International Solid-State Circuits Conference, ISSCC 2016, Digest of Technical Papers*, pp. 262–263, 2016.

[25] t. C. Developers, "Chisel/firrtl: Home."

[26] "Veripool." https://www.veripool.org/.

[27] "Xilinx vivado." https://www.xilinx.com/products/design-tools/vivado.html. Accessed: 2019-11-27.

[28] "Xilinx vcu118 fpga board." https://www.xilinx.com/products/boards-and-kits/vcu118.html. Accessed: 2019-11-27.

[29] A. Samajdar, Y. Zhu, P. Whatmough, M. Mattina, and T. Krishna, "Scale-sim: Systolic cnn accelerator simulator," *arXiv preprint arXiv:1811.02883*, 2018.

[30] A. Samajdar, Y. Zhu, and P. Whatmough, "Scale-sim." https://github.com/ARM-software/SCALE-Sim, 2018.

[31] A. Krizhevsky, "Learning multiple layers of features from tiny images," tech. rep., 2009.

[32] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, (USA), pp. 1097–1105, Curran Associates Inc., 2012.

[33] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *3rd International Conference on Learning Representations*, 2015.

[34] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," 2017. cite arxiv:1704.04861.

[35] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "ImageNet: A Large-Scale Hierarchical Image Database," in *CVPR09*, 2009.

[36] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, and O. Temam, "Dadiannao: A machine-learning supercomputer," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, 2014.

[37] V. Gokhale, J. Jin, A. Dundar, B. Martini, and E. Culurciello, "A 240 g-ops/s mobile coprocessor for deep neural networks," in *2014 IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pp. 696–701, 2014.

[38] M. Sankaradas, V. Jakkula, S. Cadambi, S. Chakradhar, I. Durdanovic, E. Cosatto, and H. P. Graf, "A massively parallel coprocessor for convolutional neural networks," in *2009 20th IEEE International Conference on Application-specific Systems, Architectures and Processors*, pp. 53–60, 2009.

[39] L. Cavigelli and L. Benini, "Origami: A 803-gop/s/w convolutional network accelerator," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 27, no. 11, pp. 2461–2475, 2017.

[40] Z. Du, R. Fasthuber, T. Chen, P. Ienne, L. Li, T. Luo, X. Feng, Y. Chen, and O. Temam, "Shidiannao: Shifting vision processing closer to the sensor," in *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, pp. 92–104, 2015.

[41] V. Sze, Y. Chen, J. Emer, A. Suleiman, and Z. Zhang, "Hardware for machine learning: Challenges and opportunities," in *2017 IEEE Custom Integrated Circuits Conference (CICC)*, pp. 1–8, 2017.

[42] P. Gysel and S. Ghiasi, "Hardware-oriented approximation of convolutional neural networks," 04 2016.

[43] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon, "In-datacenter performance analysis of a tensor processing unit," in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pp. 1–12, 2017.

[44] R. Andri, L. Cavigelli, D. Rossi, and L. Benini, "Yodann: An architecture for ultralow power binary-weight cnn acceleration," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 1, pp. 48–60, 2018.

[45] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, "Xnor-net: Imagenet classification using binary convolutional neural networks," 2016.

[46] M. Buckler, P. Bedoukian, S. Jayasuriya, and A. Sampson, "Eva2: Exploiting temporal redundancy in live computer vision," in *Proceedings of the 45th Annual International Symposium on Computer Architecture*, 2018.