Energy Efficient Boosting of GEMM Accelerators for DNN via Reuse

NIHAT MERT CICEK, Aselsan Corp., Turkey
XIPENG SHEN, North Carolina State University, USA
OZCAN OZTURK, Bilkent University, Turkey

Reuse-centric convolutional neural networks (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 article presents an in-depth exploration of architectural support for reuse-centric CNN. It addresses some major limitations of the state-of-the-art design and proposes a novel hardware accelerator that 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 neural 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 7.75× in performance. Furthermore, it can reduce the energy used for similarity detection up to 95.46%, and it can accelerate the convolutional layer up to 3.63× compared to the software-based implementation running on the CPU.

CCS Concepts: • **Computer systems organization → Special purpose systems**; **Heterogeneous (hybrid) systems**; *Multicore architectures;*

Additional Key Words and Phrases: Reuse, deep neural networks, gemm, accelerator

ACM Reference format:
Nihat Mert Cicek, Xipeng Shen, and Ozcan Ozturk. 2022. Energy Efficient Boosting of GEMM Accelerators for DNN via Reuse. *ACM Trans. Des. Autom. Electron. Syst.* 27, 5, Article 43 (June 2022), 26 pages.
https://doi.org/10.1145/3503469

1 INTRODUCTION

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. Convolution is a common method to extract information from images. Therefore, **convolutional neural networks (CNN)** have been widely used in many data mining and machine

This work has been supported in part by a grant by a grant from the Presidency of the Republic of Turkey Presidency of Defence Industries (SSB) and Aselsan A.S., with the researcher training program for defence industry (SAYP), and a grant from Technological Research Council of Turkey (TUBITAK) 1001 program through the EEEAG 119E559 project.
Authors' addresses: N. M. Cicek, Aselsan Corp., Mehmet Akif Ersoy Mahallesi Istiklal Marsi Caddesi No: 16, 06200 Yenimahalle, Ankara, Turkey; email: nmcicek@aselsan.com.tr; X. Shen, Department of Computer Science, College of Engineering, 890 Oval Drive, Engineering Building II, Raleigh, NC 27695, USA; email: xshen5@ncsu.edu; O. Ozturk, Computer Engineering Department, Bilkent University, Ankara, Turkey; email: ozturk@cs.bilkent.edu.tr.

1084-4309/2022/06-ART43 $15.00
https://doi.org/10.1145/3503469

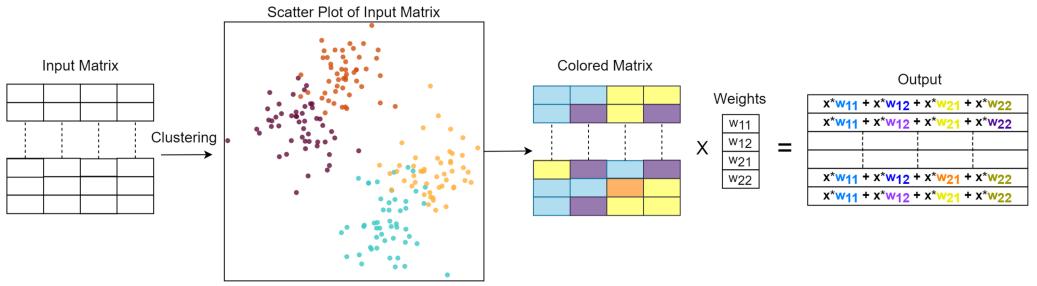

Fig. 1. An example matrix-multiplication based convolution operation leveraging general reuse. Color coding shows the available reuse.

learning domains in recent years. They are the pillars supporting many important tasks, from object recognition to autonomous driving, gesture recognition, and so on.

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 powerful compute nodes such as GPU or TPU. However, 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, and the fully connected layer [50]. Among them, the most time-consuming one is the convolutional layer [15]. Therefore, many researchers focus on accelerating the convolutional layer during inference [11, 24, 46, 47, 52].

Prior efforts on optimizing CNN inference efficiency largely fall into three categories. Some explore different convolution algorithms [33, 35], some compress networks [25], and most of all optimize matrix multiplications, the core operation in CNN, which mainly resorts to the leverage of network sparsity [24, 52], memory layout [21], and special hardware units [26, 42, 46, 47].

As a approach complementary to these efforts, reusing similar neuron vectors (sequence of consecutive neurons in a convolutional layer) has been explored in the literature [37, 38]. When the method is applied to a matrix-matrix multiplication– (GEMM) based implementation of CNN, it first uses online clustering to reduce the original data matrix to a much smaller centroid matrix, then uses the centroid matrix to conduct GEMM with the weight matrix, and finally recovers the full output matrix through value duplication.

An example of a one-dimensional convolution using this method is illustrated in Figure 1. In this figure, similar neuron vectors in the unfolded input matrix are clustered in the same color. Every four elements in each row of the unfolded image correspond to the value of a neuron vector. Based on neuron vector similarities, four clusters are formed, represented in different colors. Then, only centroid of each cluster is multiplied with an unfolded weight matrix $\mathbf{W}$, producing an output matrix $\mathbf{y}$ such that $\mathbf{y} = \mathbf{X} \cdot \mathbf{W}$. 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., $\mathbf{x}_{12} \cdot \mathbf{w}_{11}$).

For this idea to work, three conditions must hold: (1) there should be enough similarities among pixels; (2) when reused, loss in accuracy should be negligible; and (3) overhead due to similarity detection should be low. Ning and others [37, 38] state that there exists a wide set of neuron vector similarities in commonly used deep learning workloads, satisfying the first condition. In addition, they guarantee that loss in accuracy is negligible, eliminating the second condition. However, the third condition in their implementation is rather hard to achieve, because software implementation suffers from increased execution time and energy consumption due to similarity detection. In this

article, we propose hardware solutions to solve this problem especially from an energy reduction perspective.

In this work, we extend our previously proposed hardware architecture [14] for DNN implementation. The existing design had several major limitations, which we try to address by revisiting the design principles of the reuse-centric architecture. By overhauling the entire design to systematically treat both computation and memory, we create the first generalized reuse-centric DNN accelerator. This framework (i) enables reuse-centric optimizations to both computation and memory, (ii) accelerates the end-to-end processing, (iii) fits various processors (CPU, GPU, TPU, etc.), (iv) gives an order of magnitude more efficient reuse detection, (v) yields about 8× speedups for the end-to-end execution over state-of-the-art reuse-centric architecture, and (vi) up to 95.46% energy reduction for similarity detection over the software-based reuse-centric implementation.

More specifically, we propose a scalable and flexible architecture template for general reuse. The platform used for implementation is composed of a processor and hardware accelerator. A software program on the host processor initiates reuse-centric convolution operation by configuring the engine for layer-specific parameters. The accelerator is composed of four modules. The first one is the fetch unit, which is used to load and reshape data according to the convolutional layer's settings. The second one is the similarity detection unit, which clusters similar input vectors. The third one is the dispatch unit, which sends processed neuron vectors to the load store unit for storage. Finally, the load store unit computes and stores the cluster centroids. In this architecture, parallelism is provided in two dimensions: processing different neuron vectors in the same cycle and using multiple hardware. While not limited to, our accelerator can be used on the emerging edge devices that require DNN operations such as autonomous driving. For comparison purposes, candidate DNN algorithms are also implemented in software. We show that our accelerator framework can be used to implement energy-efficient DNN acceleration with performance improvements for the edge devices. This way, the programmer or the hardware designer does not need to engage in the complicated DNN optimizations, typically required to generate an efficient architecture. To summarize, the main contributions of this work are as follows:

- We propose a framework to perform convolution operations with high performance and less energy. The accelerator detects similarities among neuron vectors to improve performance and reduce energy. Parallel processing and storage of neuron vectors boost up the performance while sparse data elimination and high resource utilization provide energy efficiency.
- We implement an accelerator-based design that can conveniently be used by various processors or accelerators such as CPU, GPU, TPU, and so on. While it can be used on any existing architecture, it is specifically targeted for inference on edge devices.
- The proposed architecture is designed in a *flexible and scalable* manner. A novel fetch unit enables flexibility, which provides compatibility with any kind of convolutional network. Scalability is provided through fine-grained, pipelined, and customizable architecture template, which enables integration into any existing platform.
- We compare our design against alternative software and hardware platforms to show limitations and difficulties in implementing a high-performance energy-efficient DNN accelerator.

In the rest of this article, we first briefly outline the overall design considerations in Section 2. Next, we describe the operations required for the design of the *general reuse discovery engine* in Section 3. In addition, we give the advanced features of the accelerator in addition to how it can be integrated into existing software and hardware CNN implementations. We discuss our experimental evaluation in Section 4, and finally we conclude.

2 OVERALL DESIGN CONSIDERATIONS

In this section, 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 given in Algorithm 1. Any convolution operation with various parameters, such as kernel size and stride, can be expressed as a matrix multiplication. This process is called as unfolding. First, the input activations are unfolded, converted to matrix form, according to the shape of the weight matrix, stride, and padding settings of the convolutional layer. Then, we use our accelerator to find the similar neuron vectors and reduce the matrix size to save computations. For each vector in this matrix, a dot product operation with a randomly filled hash table is performed to determine cluster keys. After that, centroid information for the cluster keys is updated and saved into memory. Then, matrix multiplication is performed between cluster centroids and unfolded weight matrix. Finally, convolution output is derived by replicating partial results corresponding to the original neuron vectors.

ALGORITHM 1: Matrix Multiplication Based General Reuse Centric Convolution Operation

Input: input matrix $\mathbf{x}$ with dimension $N \times M$; weight matrix $\mathbf{w}$ with dimension $K \times L$; hashing matrix $\mathbf{h}$ with dimension $H \times T$; output matrix $\mathbf{o}$ with dimension $N \times M$.
Algorithm:
Define function im2col [1] = rearrange image blocks to column matrix according to the kernel size
Unfolded input matrix: im2col($\mathbf{x}$) = x_u = $(N*M)\times(K*L)$ and unfolded weight matrix: im2col($\mathbf{w}$) = w_u = $(K*L)\times 1$ with the same stride and padding. Initialize with $ID_{vectors} = \{\}, CL_{centroids} = \{\}, Centroid_{MM} = \{\}$.
for each row vector v in x_u **do**
    Calculate dot product $dp = v \cdot h$
    $key = 0_b$ with dimension H
    for each product p in dp **do**
        if p > 0 **then**
            $key = Concatenate(1_b, key)$
        else
            $key = Concatenate(0_b, key)$
    $ID_{vectors}(v) = key$
    $CL_{centroids}(key) = weighted\_average(v, centroid_{old})$
$Centroid_{MM} = CL_{centroids} \times w_u$
for each row vector index i in x_u **do**
    $key = ID_{vectors}(i)$
    $o_i = Centroid_{MM}(key)$
return o

The execution time breakdown for a software-based general reuse-centric implementation running on a general-purpose CPU is given in Figure 2 [36]. Except for centroid matrix multiplication, the similarity detection, and full result derivation take most of the total execution time for CifarNet and AlexNet. Thus, they are the bottleneck of this acceleration method. Our motivation is to speed up the whole process by executing these steps in hardware. It is important to note that our hardware accelerator serves as an auxiliary unit and does not contain any hardware for matrix multiplication. We consider using multiplication units that already exist in the host.

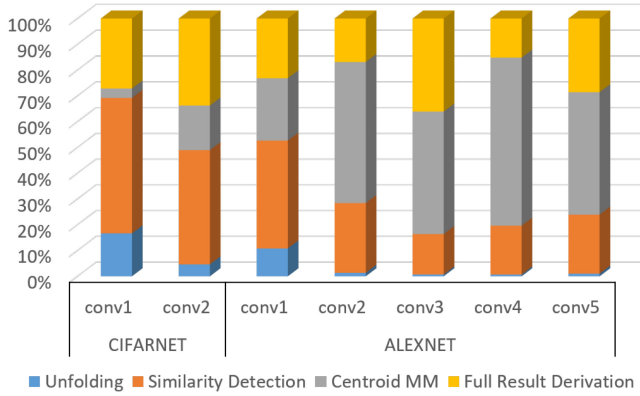

Fig. 2. The execution time breakdown of the general reuse-centric convolution operation. [36].

Key Requirements. The following principles are taken into consideration throughout the proposed design for achieving high performance and energy efficiency.

- *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 to be used in various execution environments and applications.
- *Efficiency:* The accelerator should focus on utilizing all resources effectively. This efficiency is required for both energy and performance.

Hardware Accelerator Design. Based on the breakdowns given in Figure 2, the most time-consuming blocks are similarity detection, matrix multiplication, and full result derivation. Since many existing accelerators already have efficient matrix multiplication stages, we focus on leveraging similarity detection and full result derivation. This way, we aim to increase performance and decrease the energy consumption of existing accelerators by integrating this engine.

Our design has several parameters 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. It can be reconfigured online based on the input vector size and hash vector length. Therefore, our proposed architecture can accommodate any DNN workload expressed in matrix-matrix multiplication form. 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.

3 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 section starts with an explanation of the details of the operations involved in *similarity detection* and then describes the base architecture of the accelerator.

3.1 Overview

Similarity discovery can be applied to the entire input matrix or to smaller input submatrices. As specified in Reference [38], the clustering scope determines which of the two options is more

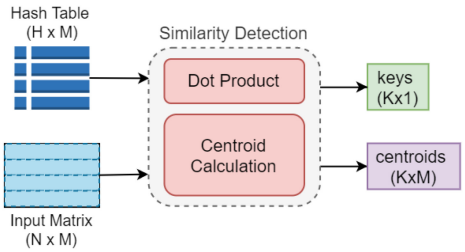

Fig. 3. Illustration of our accelerator in a similarity detection module.

effective. When clustering is performed within a single 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, that is, 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 Reference [38] 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. As 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 to extract similarities [5, 6, 16, 28, 51]. 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 3. 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.

3.2 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.

The host starts the execution by interacting with the controller inside the accelerator. Then, the controller issues signal required to control overall dataflow 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

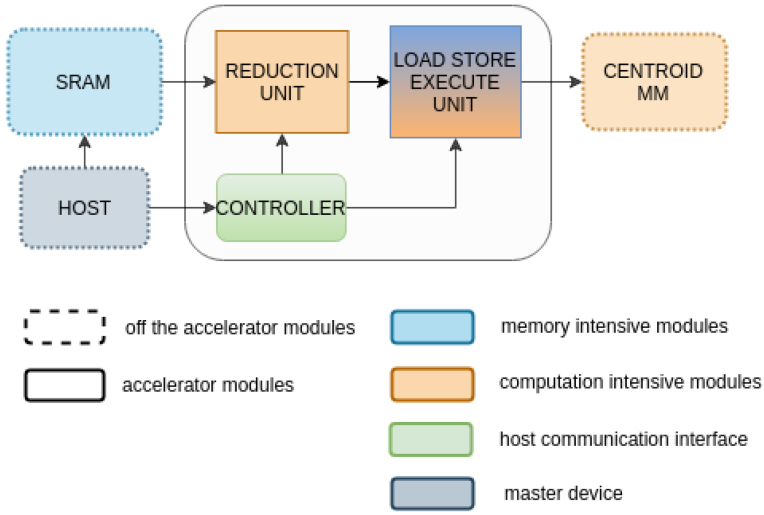

Fig. 4. Base reuse-centric accelerator architecture.

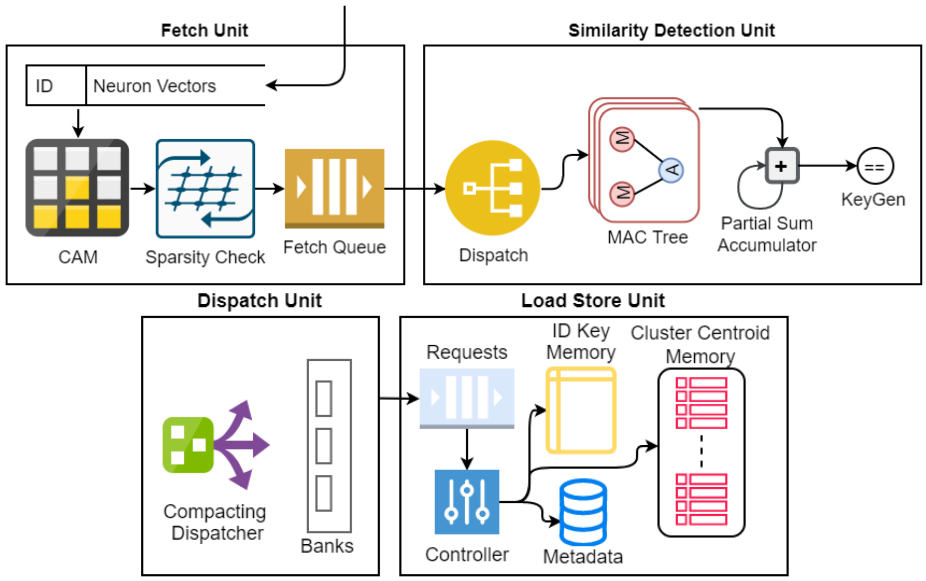

Fig. 5. Advanced accelerator design with fetch unit, similarity detection unit, dispatch unit, and load-store unit.

and fed to LSEU for updating the cluster centroid. The details of each block are described in the following subsections.

3.2.1 Reduction Unit. 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 3.1.

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.

3.2.2 Load-Store-Execute Unit. 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*.

3.2.3 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 to complete the convolution operation. Due to its simple read-write interface, it can be integrated into any host interface easily.

3.2.4 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. However, 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 Section 2, flexibility and efficiency, respectively. For this purpose, a fetch unit can be added to the design, as will be described in Section 3.3. 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. However, 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 Section 2, 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.

3.3 Hardware Implementation

Different convolutional layers have different properties in terms of neuron vector size and hash size. Therefore, our accelerator needs to support the online reconfiguration of the network-related parameters. In addition, our architecture needs to be able to process and store different sized neuron vectors in parallel for high performance and effective resource utilization. By designing a special fetch unit, similarity detection unit, dispatch unit, and load-store unit with custom tables, we aim to meet those requirements. In the rest of this section, we will explain each unit in detail.

An overview of our advanced architecture and its internal components are given in Figure 5. As can be seen from this figure, overall design consists of several stages. First, the fetch unit brings in data from SRAM and processes it to create ID neuron vectors for a given layer. Then, the neuron vector is saved into a **content addressable memory (CAM)** to prevent additional dataflow along the pipeline. After that, it checks sparsity for each neuron vector and removes the ones with a zero value, which saves energy and resources greatly. Along the pipeline, these neuron vectors are used in the clustering process. We do not gain performance by skipping zeroes, but we save energy by disabling idle functional units. At the last stage of the fetch unit, neuron vector is folded if possible and written to the queue. Afterward, neuron vectors are dispatched into a **multipy and accumulate (MAC)** tree, performing dot product operation. Then, cluster key is generated for the neuron vector. In the next step, that is during dispatch, the bank number is determined for ready keys and they are dispatched to the **load-store unit (LSU)**. This unit fetches the previously-stored neuron vector from the 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 Section 2.

Our pipeline mainly consists of four stages where each stage has a queue at the end. When a pipeline stage has been stalled, other stages can continue to work. Therefore, our design can tolerate a different amount of time for different stages. During the evaluation, we observe that overall pipeline stall time is less than 1% of all time. Therefore, each stage has almost equal pipeline time. For example, it is impossible to process multiple vectors in parallel without our **similarity detection unit (SDU)**. Thus, SDU would require more time than the other stages.

3.3.1 Fetch Unit (FU). The fetch module consists of four stages, namely, ID generation, CAM allocation, sparsity check, and folding, connected in a pipelined fashion. Micro-architectural details of this module are given in Figures 6 and 7.

ID generation stage uses two tables, division, and modulo table, to calculate the ID of each neuron vector. First, these tables are filled according to the layer's vector size and SRAM row width. Then, this module automatically starts working and ID is generated using the following equations:

$$idx' = (sramrowsize + idx) \bmod vectorsize, \tag{1}$$

$$last' = last + rem - rem' + \frac{idx + sramrowsize}{vectorsize}, \tag{2}$$

$$rem' = idx' \equiv 0. \tag{3}$$

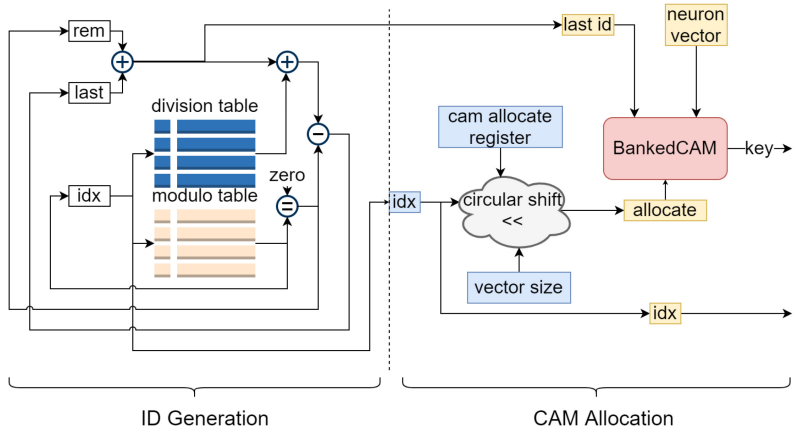

Fig. 6. Details of the fetch unit with first two stages, namely ID generation and CAM allocation.

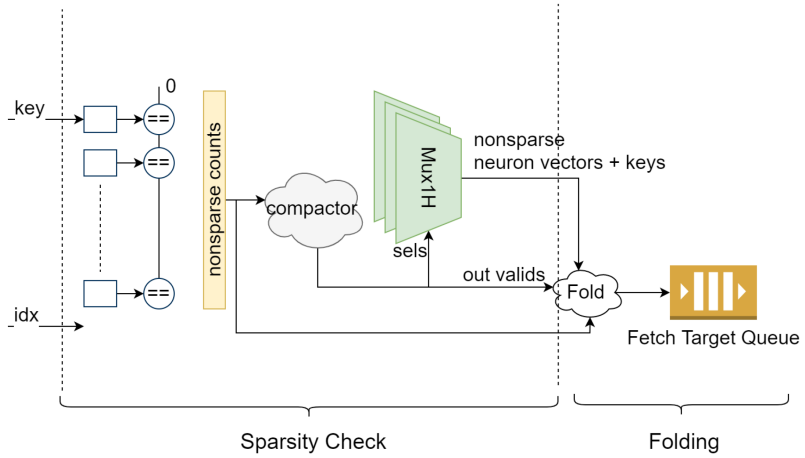

Fig. 7. Details of the fetch unit with last two stages, namely sparsity check, and queue.

In the first equation, we find the last index of the last vector in an SRAM row. In the second equation, we calculate the ID information of the last vector. In the third equation, we check whether the last vector's ID generation is finished or not. To illustrate, we give an example scenario in Figure 8 for an SRAM row size of 16 and vector size of 5. In this example, the variable "rem" and "last" is used to determine the ID of the first element in the row. The variable "idx" is used to determine where the last vector left. Thus, the ID of the first element is 9, which is equal to rem + idx. Its length is vector size, idx, which is equal to 2. The remaining of the vector is automatically filled according to the vector size. Finally, the new "rem," "idx," and "last" value are determined according to the given equation.

This stage 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 and (2) ease of integration into any existing accelerator. In addition, it also provides processing of more than one neuron vector in a single cycle. This way, it is possible to achieve parallel processing and storage, thereby, increasing performance greatly.

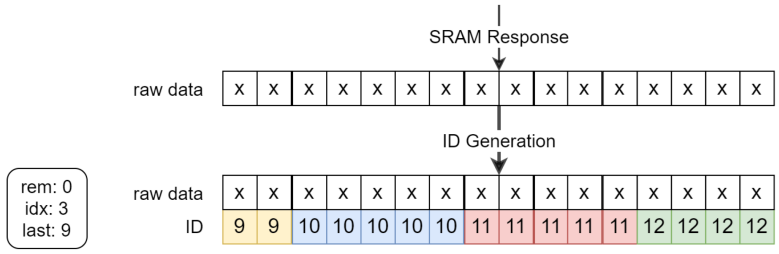

Fig. 8. Illustration of ID generation stage for an SRAM row size of 16 and vector size of 5.

After the ID generation stage, a new entry is allocated at the banked CAM for each newly generated ID by the CAM allocate logic. CAM allocate register is updated according to the layer's configuration. Then, using a circular shift register, neuron vectors and IDs are stored in CAM. Here, it is important to note that only one entry should be allocated for each ID.

To process different neuron vectors in the same cycle, banked CAM 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 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 unit imposes multiple key release requirements on deallocation logic to perform parallel processing and storage. For M banks, our deallocation logic is capable of freeing M keys during the same cycle.

In the next step, keys from CAM and neuron vectors are fed into the sparsity checker. It detects and removes zeros from each neuron vector. This way, energy is greatly saved for large sparse images and activation matrices. Finally, it is stored in fetch queue.

3.3.2 Similarity Detection Unit. The similarity detection unit performs a dot product between neuron vectors and previously-stored hash tables. It is composed of four modules, namely dispatch, MAC tree, partial sum accumulator, and key generator, as shown in Figure 5.

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

MAC tree 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 while using resources efficiently. Among many reduction trees, SIGMA's **forwarding adder network (FAN)** [40] is used because of its ability to process different IDs in the same cycle with the highest throughput possible. In this micro-architecture, shown in Figure 9, 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 IDs of neuron vectors. The algorithm used to determine control bits of multiplexers is given in Reference [40].

Although the FAN requires more resources than a regular adder tree, we prefer to enable parallel processing of neuron vectors. When we use a regular adder tree, we can perform only one dot product operation at a time. Although the FANs bring additional control and multiplexer logic, they provide parallel processing capability. Therefore, we used the FAN for high performance.

Because of the irregular data fetch and various vector size settings, a neuron vector may need to be split and processed in different cycles. Partial sum accumulator, next sub-module, checks if the dot product is completely finished for each ID. When the operation is in progress, results from

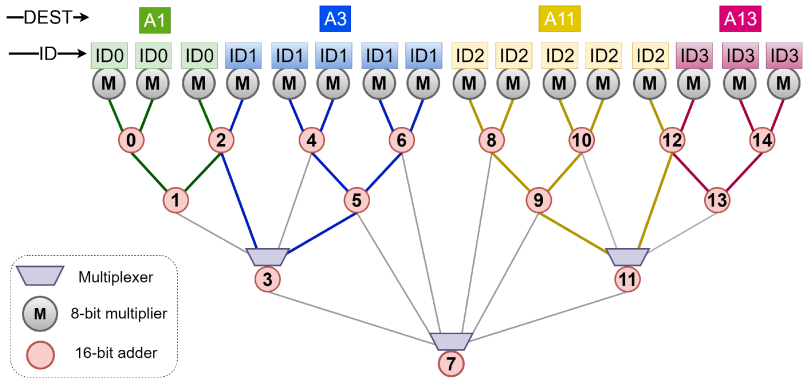

Fig. 9. Forwarding adder network topology used in our implementation [40].

the previous cycle are cumulatively stored in registers inside this module. After the dot product operation is completed, the result is used to generate a cluster key. Keygen at the end of the SDU generates the cluster key. Then, generated cluster keys along with the CAM key information are stored in a queue for bank assignment in the dispatch stage. Note that we do not need to cache cluster keys at this stage, since they propagate through the pipeline toward LSU.

Internal architecture for a forwarding adder network with 16 multipliers and 15 adders and an example execution scenario for the whole stage is given in Figure 9. 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 are 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 partial sum accumulator. Finally, the cluster key is generated using dot product results.

3.3.3 Dispatch Unit (DU). The dispatch unit performs two auxiliary functions as bank assigner and compactor, which will be explained in the following paragraphs.

As shown in Figure 10, at the end of the SDU, cluster keys are generated and written to the queues. The dispatch unit 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 $log_2 M$ bits of the cluster keys. This way we are able to get a uniform distribution of banks.

In some cases, it is possible to have more keys fetched from the SDU than the available banks in the load-store 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.

3.3.4 Load-Store Unit. This unit is designed to perform centroid calculation and storage. It contains four submodules, namely, request buffers, cluster centroid memory, ID-key memory, and controller, as shown in Figure 11.

First, dispatched keys are enqueued to their predetermined request buffers. At the same time, the neuron vector and its ID are read from the table using the associated key.

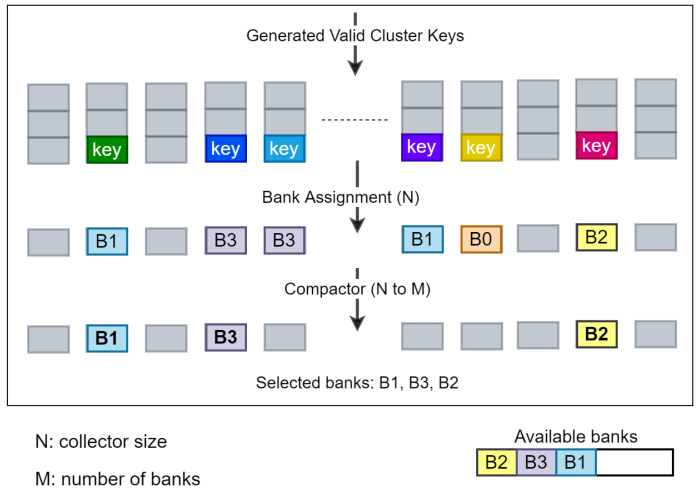

Fig. 10. Example execution scenario for the dispatch unit with N collectors and M banks.

In the next step, previous centroid information and cluster size are read from the centroid tables. In addition, metadata are read to check if the cluster has been formed before. These memories are necessary to update the cluster centroid correctly. Besides, cluster key information is written into ID-key memory for later use by the full result derivation stage mentioned in Section 2.

In the third stage, the controller updates the cluster centroid using data from previous cycles and metadata. Insertion into the centroid table 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 stalling the pipeline or forwarding updated centroids across pipeline stages, thereby, maintaining functional correctness.

To use memory efficiently, we use masking for the write operations. 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 memories helps to provide compatibility with any kind of convolutional neural networks through reconfiguration. Furthermore, banked access to all memories allows parallel execution. All these features are critical to get high performance, energy efficiency, and resource utilization.

3.3.5 Controller. The controller unit serves as the communication interface of the accelerator with the outside world. It 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, similarity detection unit, load-store unit, and banked 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.

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 to complete the convolution operation. Due to its simple read-write interface, it can be integrated into any host interface easily.

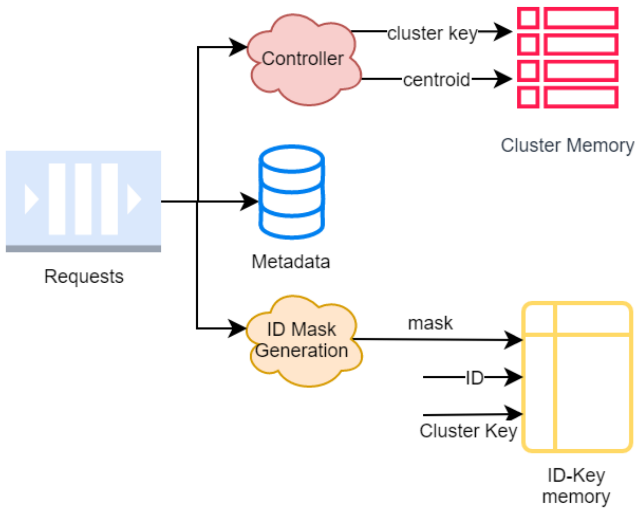

Fig. 11. The LSU is composed of hash tables to store centroids, ID-Key memory, and controller to calculate centroid and coordinate dataflow.

3.3.6 Synergy with Other CNN Accelerators. This section explains how the *general reuse discovery engine* can be integrated into an existing hardware accelerator.

In this article, we propose an architecture to speed up general matrix multiplication operation. Our architecture can be directly integrated into any GEMM accelerator. It serves as a preprocessing step and reduce the number of computations. Thus, specialized optimizations can still be utilized. However, accelerators designed for convolution operation only are out of scope for integration.

Hardware accelerators generally contain a global buffer to keep input activations, and results [11, 13, 40, 50]. 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.

An example integration into Eyeriss [11] is given in Figure 12. Eyeriss architecture [11] has a 2D mesh network consisting of global clusters, PE clusters, and router clusters. In this Eyeriss integration, our accelerator is connected to the *router cluster* to fetch raw layer inputs from the *global cluster* and send the centroid matrix to the *PE cluster* as shown in Figure 12. It is important to note that the scalability of the design enables us to fit the accelerator into any existing framework.

4 EVALUATION

To evaluate the efficiency of the proposed accelerator, we test it both with software-based CNN implementations and other CNN hardware accelerators. Energy efficiency and speedup are the major metrics used to evaluate.

4.1 Experimental Setup

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

4.1.1 Tools. We design the accelerator described in Section 3.3 using Chisel [19] hardware construction language. Furthermore, we measure the performance by running the generated RTL from

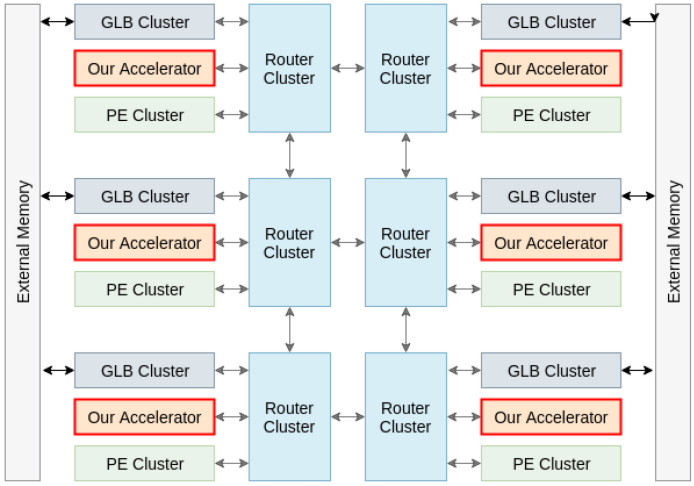

Fig. 12. Top level architecture for the integration of the reuse-centric accelerator into Eyeriss [11].

Table 1. Convolutional Network and Dataset Pairs Used in Evaluation

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

Chisel on a cycle-accurate simulator called Verilator [2]. We, then, synthesize and implement our accelerator using Xilinx's Vivado [4] tool for the VCU118 [3] evaluation board to obtain the average energy consumption and resource utilization. Additionally, to show the potential benefits of integration into existing software and hardware platforms, we use the Scale-Sim [43, 44] tool. More specifically, we measure the speedup when our accelerator is integrated into Eyeriss [13].

4.1.2 Benchmarks. We use four convolutional neural networks to evaluate our accelerator, namely, CifarNet [30], AlexNet [31], VGG-19 [48], and MobileNet [27]. Table 1 lists datasets running on described networks.

4.1.3 Datasets. We use Cifar10 [30] and Imagenet [18] as our datasets for collecting the experimental results. Cifar10 has 60,000 images with a size of 32×32 pixels, whereas the Imagenet dataset has more than 14 million images with a size of 224×224 pixels.

4.1.4 Default Parameters. The default architecture-related parameters for our advanced accelerator are given in Table 2. These optimal architectural parameters are selected for the default setup considering accuracy, performance, energy, and area.

The default CNN layer related parameters for each benchmark is given in Table 3. Note that, it is necessary to adjust the hash size and vector size online for each layer without compromising accuracy.

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

Table 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	8

(1) **CPU:** Software-based implementation running on a mobile CPU. This is used as a baseline for our experimental evaluations. This is selected as the baseline, since we would like to target our accelerator for environments where energy efficiency is critical.
(2) **Basic Accelerator:** Our basic reuse-centric architecture as explained in Section 3.1.
(3) **Advanced Accelerator:** Our advanced accelerator architecture as explained in Section 3.3.
(4) **Eyeriss (weight stationary) + Advanced Accelerator:** This is the setup where our accelerator is integrated into an existing state-of-the-art accelerator, Eyeriss [13], with weight stationary dataflow, to observe potential benefits. We use a modified version of SCALE-sim [43, 44], a CNN accelerator simulator, to measure the results obtained from this setup.

4.2 Performance and Synergy with Existing Accelerators

Previous studies [38] 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 rather focus on performance evaluation for all execution environments.

The first set of results highlight that our proposed basic and advanced accelerator implementations offer great speedups, as shown in Figure 13. 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 given in Table 2. Since VGG-19 has hash sizes greater than 16 for the first four layers, 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. Similarly to VGG-19, AlexNet has vector sizes between 10 and 24, which means that fetching a single neuron vector requires at least one cycle. However, for CifarNet and MobileNet, our advanced accelerator can process more than one neuron vector during 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 14. 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 1.46× and 3.63×. However, full result derivation and centroid multiplication still create a bottleneck.

Table 3. Algorithm Specific Parameters Used for the Benchmarks

Network	Layer Number	Batch Size	Input Vectors	Vector Size	Hash Size	Subvector
CIFARNET	conv1	100	1,024	5	15	15
CIFARNET	conv2	100	256	10	10	160
ALEXNET	conv1	100	2,916	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	50,176	9	20	3
VGG-19	conv2	16	50,176	16	20	36
VGG-19	conv3	16	12,544	16	18	36
VGG-19	conv4	16	12,544	16	18	72
VGG-19	conv5	16	3,136	16	16	72
VGG-19	conv6	16	3,136	16	16	144
VGG-19	conv7	16	3,136	16	16	144
VGG-19	conv8	16	3,136	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	12,544	18	3	9
MOBILENET	conv3	4	12,544	18	4	8
MOBILENET	conv5	4	3,136	18	4	16
MOBILENET	conv7	4	3,136	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

We further integrate our accelerator into a custom systolic array, which has the same architecture as Eyeriss but using weight stationary dataflow, to illustrate the potential benefits that can be brought by our accelerator to the existing state-of-the-art hardware accelerators. We use SCALE-sim to simulate the performance of the integration, and the execution cycles of running the entire convolutional layer on the systolic array is used as the baseline. As an integrated unit, we implement the *reuse-centric* CNN acceleration with a combination of the systolic array and our accelerator. The similarity discovery module is implemented on our accelerator while the systolic array processes the centroid matrix-multiplication. Then, host controller performs the full-result derivation by reading ID-Key memory and cluster memory to construct the final result of the convolution. Our accelerator customizes SRAMs to keep the information required for reconstruction.

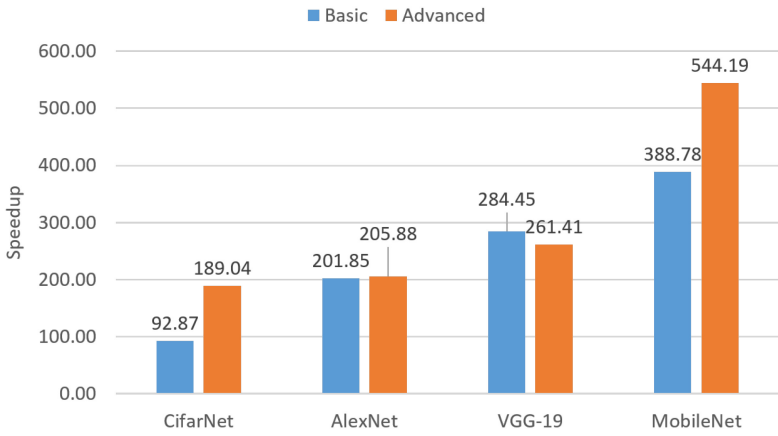

Fig. 13. Average performance improvement over mobile CPU obtained through basic and advanced design for the similarity discovery module.

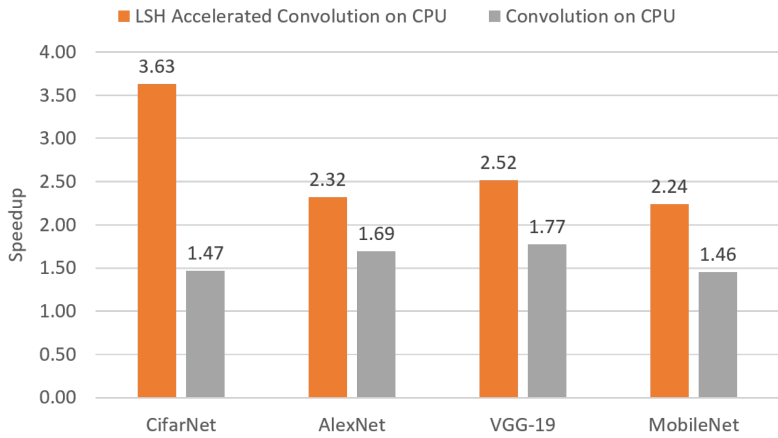

Fig. 14. Performance improvement of reuse-centric implementations on CPU and with the proposed accelerator. The baseline is the default convolution performed on a mobile CPU.

This enables the host controller to reconstruct the overall result without being affected by the memory latency. However, system architecture plays a vital role in data transfer. If the host controller does not have any cache or there is no direct memory access module in the system, then this would become the bottleneck. Therefore, selection of the embedded CPU and system design may affect the overall performance. Figure 15 illustrates the performance comparison between the baseline and the integrated unit. The results show that our proposed accelerator provides up to 7.69× speedup and very promising average speedups for all of the benchmarks, largely boosting the performance.

4.3 Energy Reduction

We perform synthesis using Xilinx's Vivado tool [4] to measure the power consumption of the advanced design on the VCU118 [3] FPGA development board. We compare our accelerator's energy consumption with the software-based implementation running on CPU as shown in Figure 16. Specifically, this figure presents the energy efficiency over mobile CPU for the similarity

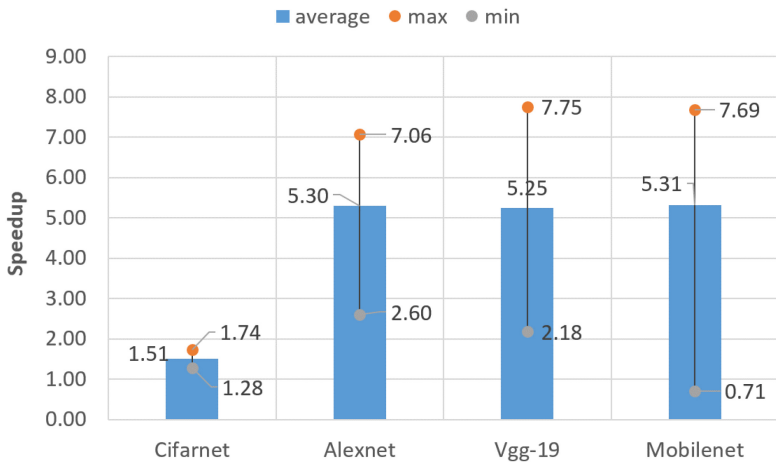

Fig. 15. The performance improvement on Eyeriss brought by our reuse-centric accelerator.

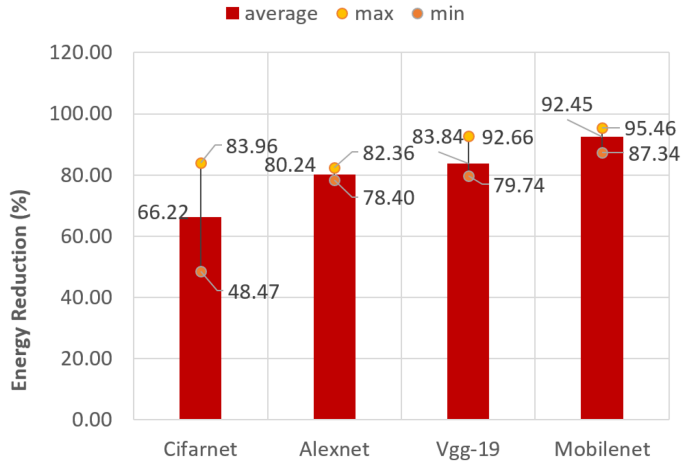

Fig. 16. Energy reduction over mobile CPU through advanced design.

discovery module. As can be seen from these results, our accelerator reduces energy consumption up to 95.46%.

4.4 Area Breakdown and Resource Utilization

Cluster memory size is too large to be implemented in an FPGA synthesis for the given parameters in Table 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 17. 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 FANs. We give specific details about the resources used by each module in Table 4.

4.5 Discussion

In the load store 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,

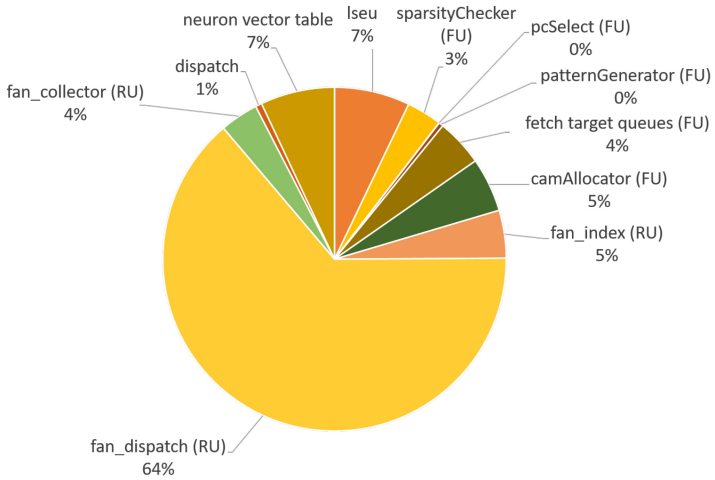

Fig. 17. Area breakdown of the modules in the advanced architecture based on the LUTs and registers.

Table 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

namely the centroid memory's actual size. However, 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 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 can use a much smaller memory together with a special hashing function to minimize collisions. Cuckoo hashing [39] for cluster centroid memories perfectly fits in this case, since it offers worst case O(1) lookup. Architecture of the load store unit with cuckoo hashing is given in Figure 18.

5 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 [10]. First, no local reuse–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 Reference [12], 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 [22], which caches the incoming weights to use during the convolution operation. Similarly, filter weights of CNN accelerators in

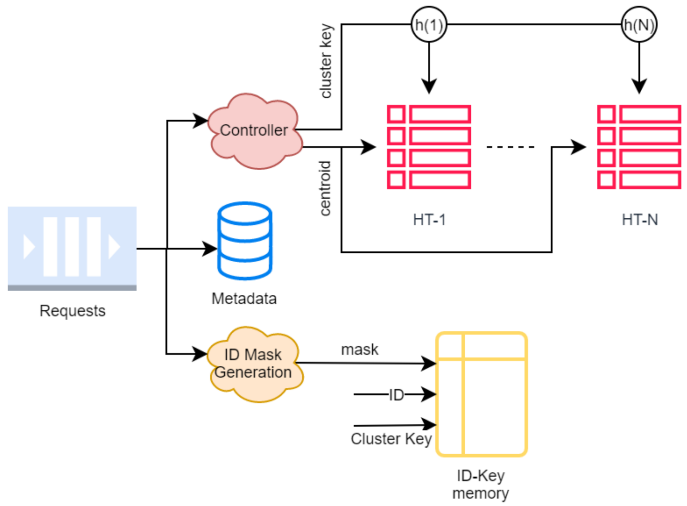

Fig. 18. Load store unit with cuckoo hashing.

References [45] and [9] 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 [20], 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 Reference [10] called row stationary, suggests maximum reuse of all types of data, namely input feature maps, filter weights, and partial sums inside a processing engine. Eyeriss [13] 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 kinds 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 [49]. 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 [23] while maintaining accuracy. An example accelerator using 8-bit integer arithmetic is Google's Tensor Processing Unit [29]. Precision can be reduced to only one or two bits as implemented in YodaNN [7] and in XNOR-Net [41]. 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 [13]. 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. In addition, **tensor train (TT)** decomposition is a common technique utilized to compress the DNN model. TIE [17] proposes an efficient

Table 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	187.8589	51.44
VGG-19	conv2	16.777216	206.8054	83.07
VGG-19	conv3	4.194304	79.9604	53.71
VGG-19	conv4	4.194304	104.6872	41.03
VGG-19	conv5	1.048576	59.9703	17.90
VGG-19	conv6	1.048576	51.2999	20.93
VGG-19	conv7	1.048576	47.2055	22.75
VGG-19	conv8	1.048576	33.4774	32.07
VGG-19	conv9	0.524288	20.6725	25.97
VGG-19	conv10	0.589824	27.3208	22.11
VGG-19	conv11	0.589824	28.4497	21.23
VGG-19	conv12	0.589824	30.2561	19.96
VGG-19	conv13	0.073728	10.4428	7.23
VGG-19	conv14	0.073728	9.0316	8.36
VGG-19	conv15	0.073728	7.7333	9.76
VGG-19	conv16	0.073728	7.056	10.70
MOBILENET	conv1	0.786432	0.75264	1069.98
MOBILENET	conv3	1.048576	0.0802	13374.69
MOBILENET	conv5	1.048576	0.2709	3962.87
MOBILENET	conv7	2.097152	4.6362	463.19
MOBILENET	conv9	0.008192	0.4164	20.14
MOBILENET	conv11	0.008192	0.7777	10.79
MOBILENET	conv13	0.008192	0.392	21.40
MOBILENET	conv15	0.008192	0.5375	15.61
MOBILENET	conv17	0.008192	0.4785	17.53
MOBILENET	conv19	0.008192	0.4108	20.42
MOBILENET	conv21	0.002048	0.2546	8.24
MOBILENET	conv23	0.002048	0.2540	8.26
MOBILENET	conv25	0.002048	0.1050	19.96
MOBILENET	conv27	0.004096	0.4045	10.37

inference scheme by eliminating redundant operations due to TT decomposition, revealing a good example in terms of hardware/software co-design perspective. Our accelerator and TIE [17] can work collaboratively to achieve higher throughput and energy efficiency.

There are also recent works on hardware accelerators leveraging computation reuse. For example, Riera et al. [42] investigate similarities between consecutive frames. Specifically, they miss the similarities across frames at different layers or batches. The study by Buckler et al. [8] 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. [26] 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, Reference [42] and Reference [8] only makes use of temporal reuse opportunities while [26] suggests spatial reuse to reduce computation and memory reads. However, our general reuse-centric CNN accelerator takes advantage of both temporal and spatial reuse for better performance and energy.

Another prominent field of study is modeling tensor dataflow to gain more insight into data reuse, bandwidth, latency, and energy consumption. Both MAESTRO [32] and TENET [34] describe frameworks that help in the analysis and evaluation of various architectures. As stated in Reference [32], there is no single hardware and dataflow for all DNN layers. Therefore, fast exploration of the huge design space is crucial to bring out the best throughput and energy.

6 CONCLUSION

In this article, we propose an architecture that reuses similar neuron vectors to boost the performance of CNN executions. Major design differences from the prior work [14] can be summarized as follows:

(1) Previous design was memory centric. Current architecture is both memory and computation centric. We have optimized computation by enabling parallel processing of neuron vectors. Thus, we designed custom hardware blocks to fetch, perform similarity detection, and store clustering results.
(2) Previous design had caches to reduce off chip data movement. However, this design uses customized banked SRAM blocks to store centroids and key information.
(3) This accelerator design is flexible, scalable, and resource-efficient to enable integration into any existing accelerator and to speed up CNN inference.

We measured its performance through four CNNs and integrated it into Eyeriss [13]. Specifically, it provides speedups up to 7.75× for a convolutional layer. Furthermore, we also integrated into a software-based implementation of general reuse-centric CNN acceleration. We observe up to 3.63× faster execution in a convolutional layer and energy reduction up to 95.46% in similarity detection for the same CNN.

In our work, the proposed field of application is convolutional neural networks. We represent convolution as a matrix multiplication. Then, we apply LSH to accelerate matrix multiplication. Therefore, any operation represented as a matrix multiplication can be accelerated using our hardware. From this perspective, other workloads such as deep learning-based recommendation systems and audio applications can also be accelerated using our framework, since they can be built using CNNs or RNNs. However, it is important to note that there are certain conditions that must hold for LSH to work efficiently for other workloads as we stated in Section 1.

REFERENCES

[1] How Are Convolutions Actually Performed Under the Hood? Retrieved September 9, 2021 from https://towardsdatascience.com/how-are-convolutions-actually-performed-under-the-hood-226523ce7fbf.
[2] Veripool. Retrieved from https://www.veripool.org/.
[3] Xilinx VCU118 FPGA Board. Retrieved November 27, 2019 from https://www.xilinx.com/products/boards-and-kits/vcu118.html.
[4] Xilinx Vivado. Retrieved November 27, 2019 from https://www.xilinx.com/products/design-tools/vivado.html.
[5] Alexandr Andoni and Piotr Indyk. 2006. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. In *Proceedings of the 47th Annual IEEE Symposium on Foundations of Computer Science (FOCS'06)*. 459–468.
[6] Alexandr Andoni, Piotr Indyk, Thijs Laarhoven, Ilya Razenshteyn, and Ludwig Schmidt. 2015. Practical and optimal LSH for angular distance. In *Proceedings of the 28th International Conference on Neural Information Processing Systems, Volume 1*. MIT Press, Cambridge, MA, 1225–1233.

[7] R. Andri, L. Cavigelli, D. Rossi, and L. Benini. 2018. YodaNN: An architecture for ultralow power binary-weight CNN acceleration. *IEEE Trans. Comput.-Aid. Des. Integr. Circ. Syst.* 37, 1 (2018), 48–60. https://doi.org/10.1109/TCAD.2017.2682138

[8] Mark Buckler, Philip Bedoukian, Suren Jayasuriya, and Adrian Sampson. 2018. EVA2: Exploiting temporal redundancy in live computer vision. In *Proceedings of the 45th Annual International Symposium on Computer Architecture*.

[9] L. Cavigelli and L. Benini. 2017. Origami: A 803-GOp/s/W convolutional network accelerator. *IEEE Trans. Circ. Syst. Vid. Technol.* 27, 11 (2017), 2461–2475. https://doi.org/10.1109/TCSVT.2016.2592330

[10] Y. Chen, J. Emer, and V. Sze. 2016. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. In *Proceedings of the ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA'16)*. 367–379. https://doi.org/10.1109/ISCA.2016.40

[11] Yu-Hsin Chen, Joel S. Emer, and Vivienne Sze. 2018. Eyeriss v2: A flexible and high-performance accelerator for emerging deep neural networks. arXiv:1807.07928. Retrieved from http://arxiv.org/abs/1807.07928.

[12] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, and Olivier Temam. 2014. DaDianNao: A machine-learning supercomputer. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*.

[13] Chen, Yu-Hsin and Krishna, Tushar and Emer, Joel and Sze, Vivienne. 2016. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. In *Proceedings of the IEEE International Solid-State Circuits Conference (ISSCC'16), Digest of Technical Papers*. 262–263.

[14] Nihat Mert Cicek, Lin Ning, Ozcan Ozturk, and Xipeng Shen. 2022. General reuse-centric CNN accelerator. *IEEE Trans. Comput.* 71, 4 (2022), 880–891. https://doi.org/10.1109/TC.2021.3064608

[15] Jason Cong and Bingjun Xiao. 2014. Minimizing computation in convolutional neural networks. In *Proceedings of the International Conference on Artificial Neural Networks and Machine Learning (ICANN'14)*, Stefan Wermter, Cornelius Weber, Włodzisław Duch, Timo Honkela, Petia Koprinkova-Hristova, Sven Magg, Günther Palm, and Alessandro E. P. Villa (Eds.). Springer International Publishing, Cham, 281–290.

[16] Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab S. Mirrokni. 2004. Locality-sensitive hashing scheme based on p-stable distributions. In *Proceedings of the 20th Annual Symposium on Computational Geometry*. ACM, New York, NY, 253–262.

[17] Chunhua Deng, Fangxuan Sun, Xuehai Qian, Jun Lin, Zhongfeng Wang, and Bo Yuan. 2019. TIE: Energy-efficient tensor train-based inference engine for deep neural network. In *Proceedings of the ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA'19)*. 264–277.

[18] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. 2009. ImageNet: A large-scale hierarchical image database. In *Proceedings of the Conference on Computer Vision and Pattern Recognition (CVPR'09)*.

[19] The Chisel/FIRRTL Developers. Chisel/FIRRTL: Home. Retrieved from https://www.chisel-lang.org/.

[20] Z. Du, R. Fasthuber, T. Chen, P. Ienne, L. Li, T. Luo, X. Feng, Y. Chen, and O. Temam. 2015. ShiDianNao: Shifting vision processing closer to the sensor. In *Proceedings of the ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA'15)*. 92–104. https://doi.org/10.1145/2749469.2750389

[21] Mingyu Gao, Jing Pu, Xuan Yang, Mark Horowitz, and Christos Kozyrakis. 2017. TETRIS: Scalable and efficient neural network acceleration with 3D memory. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'17)*. Association for Computing Machinery, New York, NY, 751–764. https://doi.org/10.1145/3037697.3037702

[22] V. Gokhale, J. Jin, A. Dundar, B. Martini, and E. Culurciello. 2014. A 240 G-ops/s Mobile coprocessor for deep neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*. 696–701. https://doi.org/10.1109/CVPRW.2014.106

[23] Philipp Gysel, Mohammad Motamedi, and Soheil Ghiasi. 2016. Hardware-oriented Approximation of Convolutional Neural Networks. arxiv:1604.03168. Retrieved from https://arxiv.org/abs/1604.03168.

[24] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A. Horowitz, and William J. Dally. 2016. EIE: Efficient inference engine on compressed deep neural network. In *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA'16)*. 243–254. https://doi.org/10.1109/ISCA.2016.30

[25] Song Han, Huizi Mao, and William J. Dally. 2016. Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding. In *Proceedings of the 4th International Conference on Learning Representations (ICLR'16)*, Yoshua Bengio and Yann LeCun (Eds.).

[26] Kartik Hegde, Jiyong Yu, Rohit Agrawal, Mengjia Yan, Michael Pellauer, and Christopher W. Fletcher. 2018. UCNN: Exploiting computational reuse in deep neural networks via weight repetition. In *Proceedings of the 45th Annual International Symposium on Computer Architecture*.

[27] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. MobileNets: Efficient convolutional neural networks for mobile vision applications. arXiv:1704.04861. Retrieved from http://arxiv.org/abs/1704.04861.

[28] Piotr Indyk and Rajeev Motwani. 1998. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *Proceedings of the 30th Annual ACM Symposium on Theory of Computing*. 604–613.

[29] 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. 2017. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA'17)*. 1–12. https://doi.org/10.1145/3079856.3080246

[30] Alex Krizhevsky. 2009. *Learning Multiple Layers of Features from Tiny Images*. Technical Report.

[31] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2012. ImageNet classification with deep convolutional neural networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems, Volume 1*. Curran Associates Inc., USA, 1097–1105.

[32] Hyoukjun Kwon, Prasanth Chatarasi, Vivek Sarkar, Tushar Krishna, Michael Pellauer, and Angshuman Parashar. 2020. MAESTRO: A data-centric approach to understand reuse, performance, and hardware cost of DNN mappings. *IEEE Micro* 40, 3 (2020), 20–29. https://doi.org/10.1109/MM.2020.2985963

[33] A. Lavin and S. Gray. 2016. Fast algorithms for convolutional neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR'16)*. 4013–4021. https://doi.org/10.1109/CVPR.2016.435

[34] Liqiang Lu, Naiqing Guan, Yuyue Wang, Liancheng Jia, Zizhang Luo, Jieming Yin, Jason Cong, and Yun Liang. 2021. TENET: A framework for modeling tensor dataflow based on relation-centric notation. In *Proceedings of the ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA'21)*. 720–733. https://doi.org/10.1109/ISCA52012.2021.00062

[35] Michael Mathieu, Mikael Henaff, and Yann LeCun. 2014. Fast training of convolutional networks through FFTS. In *Proceedings of the* 2nd International Conference on Learning Representations, *(ICLR'14)*.

[36] Lin Ning. 2020. *Deep Reuse for Deep Learning*. Ph.D. Dissertation. North Carolina State University.

[37] Lin Ning, Hui Guan, and Xipeng Shen. 2019. Adaptive deep reuse: Accelerating CNN training on the fly. In *Proceedings of the IEEE 35th International Conference on Data Engineering (ICDE'19)*.

[38] Lin Ning and Xipeng Shen. 2019. Deep reuse: Streamline CNN inference on the fly via coarse-grained computation reuse. In *Proceedings of the ACM International Conference on Supercomputing (ICS'19)*.

[39] Rasmus Pagh and Flemming Friche Rodler. 2004. Cuckoo hashing. *J. Algor.* 51, 2 (2004), 122–144. https://doi.org/10.1016/j.jalgor.2003.12.002

[40] E. Qin, A. Samajdar, H. Kwon, V. Nadella, S. Srinivasan, D. Das, B. Kaul, and T. Krishna. 2020. SIGMA: A sparse and irregular GEMM accelerator with flexible interconnects for DNN training. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA'20)*. 58–70. https://doi.org/10.1109/HPCA47549.2020.00015

[41] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. 2016. XNOR-Net: ImageNet classification using binary convolutional neural networks. arXiv:cs.CV/1603.05279. Retrieved from https://arxiv.org/abs/1603.05279.

[42] Marc Riera, Jose-Maria Arnau, and Antonio González. 2018. Computation reuse in DNNs by exploiting input similarity. In *Proceedings of the 45th Annual International Symposium on Computer Architecture (ISCA'18)*. IEEE Press, Piscataway, NJ, 57–68.

[43] Ananda Samajdar, Yuhao Zhu, and Paul Whatmough. 2018. SCALE-Sim. Retrieved from https://github.com/ARM-software/SCALE-Sim.

[44] Ananda Samajdar, Yuhao Zhu, Paul Whatmough, Matthew Mattina, and Tushar Krishna. 2018. SCALE-Sim: Systolic CNN accelerator simulator. arXiv:1811.02883. Retrieved from https://arxiv.org/abs/1811.02883.

[45] M. Sankaradas, V. Jakkula, S. Cadambi, S. Chakradhar, I. Durdanovic, E. Cosatto, and H. P. Graf. 2009. A massively parallel coprocessor for convolutional neural networks. In *Proceedings of the 20th IEEE International Conference on Application-specific Systems, Architectures and Processors*. 53–60. https://doi.org/10.1109/ASAP.2009.25

[46] H. Sharma, J. Park, D. Mahajan, E. Amaro, J. K. Kim, C. Shao, A. Mishra, and H. Esmaeilzadeh. 2016. From high-level deep neural models to FPGAs. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'16)*.

[47] Yongming Shen, Michael Ferdman, and Peter Milder. 2017. Maximizing CNN accelerator efficiency through resource partitioning. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*.

[48] Karen Simonyan and Andrew Zisserman. 2015. Very deep convolutional networks for large-scale image recognition. In *Proceedings of the 3rd International Conference on Learning Representations*.

[49] V. Sze, Y. Chen, J. Emer, A. Suleiman, and Z. Zhang. 2017. Hardware for machine learning: Challenges and opportunities. In *Proceedings of the IEEE Custom Integrated Circuits Conference (CICC'17)*. 1–8. https://doi.org/10.1109/CICC.2017.7993626

[50] V. Sze, Y. Chen, T. Yang, and J. S. Emer. 2017. Efficient processing of deep neural networks: A tutorial and survey. *Proc. IEEE* 105, 12 (December 2017), 2295–2329. https://doi.org/10.1109/JPROC.2017.2761740

[51] Kengo Terasawa and Yuzuru Tanaka. 2007. Spherical lsh for approximate nearest neighbor search on unit hypersphere. In *Proceedings of the Workshop on Algorithms and Data Structures*. 27–38.

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

Received May 2021; revised November 2021; accepted November 2021