General Reuse-Centric CNN Accelerator

Nihat Mert Cicek, *Member,* Lin Ning, *Member,*
Ozcan Ozturk, *Senior Member,* and Xipeng Shen, *Senior Member*

Abstract—This paper introduces the first *general reuse-centric accelerator* for CNN inferences. Unlike prior work that exploits similarities only across consecutive video frames, *general reuse-centric accelerator* is able to discover similarities among arbitrary patches within an image or across independent images, and translate them into computation time and energy savings. Experiments show that the accelerator complements both prior software-based CNN and various CNN hardware accelerators, producing up to 14.96X speedups for similarity discovery, up to 3.33X speedups for a convolutional layer.

Index Terms—CNN, reuse-centric, accelerator

1 Introduction

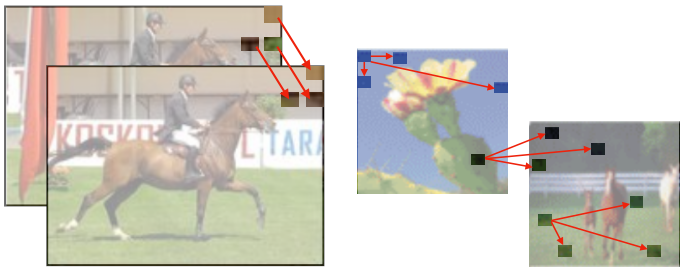
Convolutional Neural Networks (CNN) have been widely used in many data mining and machine learning domains in recent years. They are the pillars supporting many important tasks, from object recognition, to autonomous driving, gesture recognition, and so on. The inference speed and energy efficiency of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

The needs have driven a strand of efforts in developing special hardware accelerators [1]–[4]. These accelerators have mostly centered around streamlining matrix multiplication, the core computation in CNN.

A complementary direction is exploration of computation reuse in CNN accelerators to save energy and time. This direction has been explored only recently. The existing explorations [2], [5], [6] have been mostly focused on temporal similarities of corresponding elements (e.g., pixels or objects) across consecutive video or speech frames. In a recent work [2], for instance, the authors avoid the computation on a pixel if it (after quantization) stays unchanged from the previous frame. We call these designs *temporal reuse-based accelerators*.

In this work, we propose *general reuse-centric* CNN accelerator. It features a much more general kind of reuse. As Figure 1 (a) shows, the traditional reuse is limited to between *corresponding* elements in *consecutive* frames only. *General reuse*, on the other hand, can be between elements from two *different* objects in the same image (e.g., the sky patches in the first image, grass patches in the second image in Figure 1 (b)); it can also be between elements in two totally independent images (e.g., the patch on the cactus in the first image and the dark forest in the second image in Figure 1 (b)). In addition, in *general reuse*, the reuse can have a flexible granularity, from as small as one element to as large as a large tile.

In a sense, the previously explored *temporal reuse* can be regarded as a special case of *general reuse*: the granularity is one and limited to temporal level across frames. *General reuse* covers both *spatial similarity* and *temporal similarity* with flexible granularity.



(a) Temporal Reuse (b) General Reuse

Fig. 1. Illustration of previously explored *temporal reuse* and our proposed *general reuse*.

This generalization significantly expands the applicability of reuse-based acceleration. Many datasets—such as a collection of static images—are not temporal sequences. For such datasets, pixel-level temporal similarity is scarce. Besides better applicability, *general reuse* can magnify the benefits of reuse for temporal sequence data by uncovering more acceleration opportunities.

We draw on Figure 2 to explain how *general reuse* saves computations in CNN. We first introduce a term, *neuron vector*, which refers to a sequence of consecutive neurons in an input layer or a middle layer—*general reuse* applies also to the data (called *activation maps*) on middle layers. In Figure 2, $\mathbf{x}$ is an unfolded input matrix. Each element corresponds to the value of one neuron. Every three elements in the example correspond to the value of a neuron vector. Based on neuron vector similarities, the eight neuron vectors fall into four groups, represented in four 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., $\mathbf{x}_{13} \cdot \mathbf{w}_{11}$ and $\mathbf{x}_{14} \cdot \mathbf{w}_{11}$).

General reuse-centric CNN accelerator is a hardware accelerator built on *general reuse*. It is designed to effectively translate neuron vector similarities into computation time savings.

The generality of *general reuse* entails a series of special challenges in creating the accelerator.

- *N.M. Cicek and O. Ozturk are with Department of Computer Engineering, Bilkent University, Turkey.*
  E-mail: nihat.cicek@bilkent.edu.tr, ozturk@cs.bilkent.edu.tr
- *L. Ning and X. Shen are with Department of Computer Science, North Carolina State University, US.*
  E-mail: xshen5@ncsu.edu, lning@ncsu.edu

(1) First, how to make it efficiently uncover *general similarities*. Unlike *temporal similarities*, a simple comparison of same-index pixels is insufficient for uncovering *general similarities* as similar neuron vectors could appear anywhere. We address the problem by borrowing the idea of locality-sensitive hashing (LSH) [7], [8] and creating a hardware-based online clustering engine.

(2) Second, how to make its design fit reuse-centric CNN computations. The computation and data access patterns that the accelerator needs to handle are different from those in regular matrix multiplications. To ensure the efficiency of the computations, we propose a tri-module coordinated hardware pipeline, assisted with a data-discerning caching scheme and an on-the-fly matrix unfolding unit to minimize both the number of memory accesses and data movements to the accelerator.

(3) Third, how to make the design able to easily adapt to data differences and layer differences. When *general reuse* is used, the right settings for the neuron vector lengths and clustering engines could differ across layers of different widths and data of different sizes. As a CNN usually consists of a stack of different layers, fast adaptations in hardware is essential. We address the challenge through a design of template-based lightweight reconfigurations.

(4) Finally, how to make *general reuse* easily integratable into other CNN accelerators. *General reuse* is an optimization complementing the prior hardware efforts in streamlining matrix multiplications. It is hence desirable if the *reuse-centric accelerator* can be made easily integratable into existing CNN accelerators to realize the compound benefits. Towards that goal, our design features a modular design, wrapping up the essential components (for similarity uncovering) into a module that can be easily plugged into other CNN accelerators.

To the best of our knowledge, this is the first work that introduces *general reuse* into CNN hardware acceleration, creates a *general reuse-centric CNN accelerator*, and demonstrates the significant benefits and synergy with existing CNN accelerators. As will be demonstrated in Section 8, it could speed up the inference stage around an order of magnitude.

2 Background

CNN As a deep neural network, a typical CNN consists of many layers, including the convolutional layer, the ReLU layer, the pooling layer and so on. The CNN can take an image as an input and output a class label for the image. Inside a CNN, each layer has its own input (e.g., input image for the first convolutional layer and activation maps for the following convolutional layers) and its output will be fed as the input to the layer next to it. Among all different types of layers, the convolutional layer is the most computationally intensive one. It usually consumes a large portion of the computation resources during the inference.

A widely used method, the *matrix multiplication-based method*, casts the inputs into matrices and leverage the high performance matrix-matrix multiplication routine of the Basic Linear Algebra Subprogram (BLAS) [10] for computation. Using the first convolutional layer as an example, an input image is unfolded into a large input matrix $\mathbf{x}$.

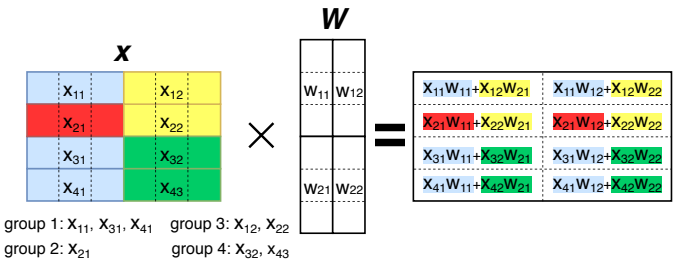


Fig. 2. An example of the basic form of computation reuse across neuron vectors in convolution $X \times W$ [9].

The input matrix is multiplied with a weight matrix $\mathbf{W}$, producing an output matrix such as $\mathbf{y} = \mathbf{x} \cdot \mathbf{W}$. Figure 2 gives an illustration of this matrix multiplication process. Let N be the number of rows of the input matrix $\mathbf{x}$, K be the number of columns of $\mathbf{x}$ and M be the number of columns of the weight matrix $\mathbf{W}$, the computation complexity of the convolutional layer is $O(N \cdot K \cdot M)$.

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.

Existence of* general reuse *opportunities Neuron vector similarities have been observed before. Ning and others have reported a broad existence of neuron similarities in popular image datasets [9], [11]. Table 1 gives some samples of the observations from the previous study: 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.

TABLE 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

Despite the observed existence of *general reuse*, there has been no way to leverage it efficiently in hardware CNN accelerators. The previous explorations [9], [11] have only used software methods to demonstrate the potential of *general reuse* for saving CNN computations. Due to lack of efficient solutions to uncover the reuse opportunities, the software methods suffer large runtime overhead (over 60% for CifarNet and 18% - 53% for AlexNet). How to effectively tap into the full potential of *general reuse* remains an open question.

3 Overall Design Considerations

This section first presents our principles in designing the *general reuse-centric accelerator*. It then describes the overall work flow of general reuse-centric convolution, and reveals the performance bottleneck by showing profiling data. After that, it gives a high-level overview of our design.

Design Principles When designing the accelerator, we follow three principles.

- *Reusability.* The accelerator should be usable for both software CNN and various hardware CNN accelerators. As reuse is complementary to other optimizations, this principle will ensure that the potential benefits of the accelerator can be maximized by working with other software and hardware artifacts.
- *Resource efficiency.* The design should focus resource in alleviating the bottleneck. This ensures high efficiency in area and cost, which is essential for it to integrate into other accelerators.
- *Memory efficiency.* The design should minimize data movements and memory usage; both are important for the efficiency and scalability of an accelerator.

Overall Work Flow and Bottleneck

To help explain the performance bottleneck in reuse-centric convolution, based on a prior work [9], we outline in Figure 3 the workflow of software-based convolution that tries to leverage neuron vector similarities. There are four main computation modules: unfolding, similarity discovery, centroid matrix-multiplication and full-result derivation. *Unfolding* is to derive a large matrix from an input image by extending each convolution window into one row in the matrix; one pixel in the image is typically duplicated on multiple rows in the unfolded matrix as the convolution tiles often have overlaps. *Similarity discovery* puts similar neuron vectors into one cluster. *Centroid matrix-multiplication (MM)* multiplies the centroids of the clusters with the weight matrix. *Full-result derivation* recovers the full output matrix by duplicating elements in the output of *centroid MM*. Many computations can be saved as *centroid MM* replaces the original unfolded input and weight matrix multiplication.

Table 2 gives the time breakdown of the modules in convolution, measured on a Huawei SE Mate mobile phone.

TABLE 2
The breakdown of the unfolding-based reuse-centric convolution running time. S.D. is similarity discover. C.MM is centroid matrix-multiplication. F.D. is full-result derivation.

Network	Layer	Unfold	S.D.	C.MM	F.D.
CifarNet	conv1	16.65%	52.56%	3.63%	27.16%
	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%	16.87%
	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%

Overview of the Design Among the main modules, we choose to concentrate our accelerator design on *general reuse discovery*, for several reasons. First, it aligns with the second principle listed at the beginning of this section. *Similarity discovery* is one of the most time consuming components in the convolution as Table 2 shows. The other time consuming component is *centroid MM*. But as it is basically a regular Matrix Multiplication, existing accelerators can already accelerate it. Second, it aligns with the first design principle as well. Specifically, we create a *general reuse discovery engine*, which discovers the similarities of neuron vectors and exposes the reuse opportunities. Compared to an entire convolution accelerator, such an engine offers much better reusability. It can be easily integrated into other convolution accelerators or software-based convolutions, meeting the first principle.

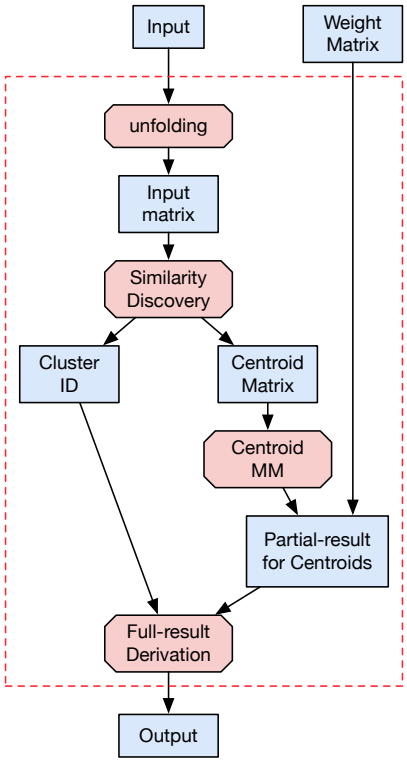


Fig. 3. Illustration of the main workflow in a similarity detection module.

In addition, we create *on-the-fly unfolding unit* to help avoid the *unfolding* overhead as well as minimizing data movements to the hardware accelerator. Along with the *data-discerning caching* feature, it minimizes data movements and memory usage, meeting the third design principle.

In the rest of this paper, we first describe the basic design of the *general reuse discovery engine* in Section 4 and its advanced features in Section 5, then explain the *on-the-fly unfolding module* in Section 6, and finally illustrate how the accelerator can be integrated into existing software and hardware CNN implementations in Section 7.

4 Basic Design of General Reuse Discovery Engine

4.1 Reuse-Centric Abstraction Model

As mentioned before, our goal is to implement a general reuse discovery engine to extract the similarities between neuron vectors and exploit reuse opportunities. While our accelerator is generic enough to be used with any software or hardware platform, it is also fast, efficient, and cost effective.

Online clustering is at the center of *general reuse discovery*. While we propose a generic framework, matrix multiplication based reuse implementations can be considered as an

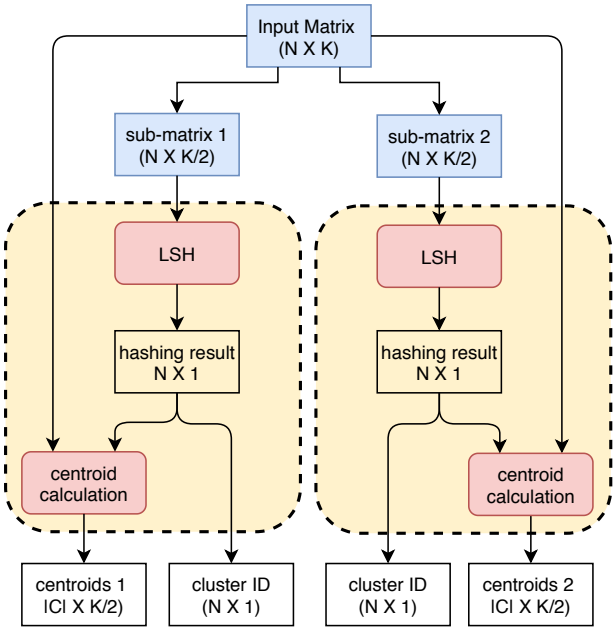


Fig. 4. Illustration of our accelerator in a similarity detection module.

example. An input matrix with a $N \times K$ size is divided into sub-matrices with shorter neuron vector lengths based on a pre-determined clustering granularity, denoted as L. For each sub-matrix, the similarity detection module identifies the similarity among neuron vectors by grouping them into clusters. It outputs the centroids for all the clusters and the cluster IDs for all the neuron vectors.

As mentioned in Section 2, there are different clustering methods for similarity detection, including K-means, HyperCube and Locality-Sensitive Hashing (LSH). Among these three methods, LSH based method can generate good clustering results for efficient neuron vector similarity identification. Meanwhile, it is light-weight and has the least overheads compared to the other two options. Therefore, LSH has been chosen for reuse-centric CNN acceleration.

Given H hashing vectors, each neuron vector in a sub-matrix is mapped to a bit vector with a length of H indicating the cluster ID of the neuron vector. Figure 4 gives an illustration of the similarity detection module. In this example, input matrix is divided into 2 sub-matrices, where $L = K/2$. Each dashed boundary represents an LSH based similarity detection module, which we accelerate through our hardware implementation.

As a method for solving the approximate nearest neighbor problem [7], [8], [12]–[14], LSH maps input vectors to bit vectors using a series of hashing functions. Specifically, for an input vector $\mathbf{x}$, a hashing function h is determined by a random vector $\mathbf{v}$ using the following formula:

$$h_{\mathbf{v}}(\mathbf{x}) = \begin{cases} 1 & if \quad \mathbf{v} \cdot \mathbf{x} > 0 \\ 0 & if \quad \mathbf{v} \cdot \mathbf{x} \leq 0 \end{cases} \tag{1}$$

Given H hashing functions, each neuron vector is mapped to a bit vector with a length of H after executing on the LSH unit. The bit vectors from all the neuron vectors in a sub-matrix form the hashing result of this sub-matrix as shown in Figure 4. Neuron vectors with small distances have high probability to be hashed to the same bit vector. Therefore, neuron vectors being mapped to the same bit vector are considered to be part of the same cluster. We use each unique bit vector in the hashing result as the corresponding cluster ID and compute the centroid of each cluster accordingly. Algorithm 1 gives a detailed description of the aforementioned LSH based similarity detection.

Algorithm 1 LSH based similarity detection

1: **Input:** input matrix $\mathbf{x}$ with dimension $N \times K$; hashing matrix $\mathbf{H}$ with dimension $K \times H$.
2: **Algorithm:**
3: Initialize with $C_{ID} = \{\ \}$, $C = \{\ \}$, $C_{count} = \{\ \}$, ID with dimension N
4: **for** each row vector x_i **do**
5: Compute the bit vectors bv
6: $ID_i = bv$
7: **if** $bv \in C_{ID}$ **then**
8: $C_{ID=bv} += x_i$
9: $C_{count,ID=bv} += 1$
10: **else**
11: $C_{ID} = C_{ID} \cup bv$
12: $C_{count,ID=bv} = 1$
13: $C_{ID=bv} = x_i$
14: **for** each id in C_{ID} **do**
15: $C_{ID=id} /= C_{count,ID=id}$
16: return C and ID

In this section, we focus the accelerator design on LSH-based similarity detection and centroid generation. In this approach, there are basic data structures corresponding to each convolutional layer along with serial functions for the aforementioned operations.

The reuse-centric framework enables the neuron vectors at a layer to be grouped into a small number of clusters, each with its own centroid. This way, the amount of vector dot product between all the neuron vectors and the weight is reduced down to the dot product between the centroids and the weight. Many deep learning models can be naturally represented using this model, thereby providing significant energy savings and speedup.

4.2 Execution Scenario

In this subsection, we describe the execution flow in the accelerator.

1) Initially, hash tables inside the LSH module are filled with random data.
2) Then, configurations are done according to the size of input feature map and network.
3) Once the configuration settings are complete, neuron vectors start streaming.
4) LSH module performs the dot product between input vectors and hash tables, generating a key (clusterID) pointing out to the cluster that neuron vector belongs to.
5) For the neuron vectors belonging to the same cluster, only centroid information is written to the memory with the index clusterID.
6) When all inputs are processed, user can request all the clusters using associated keys.

It is important to note that reconfigurability is crucial for the accelerator to fit the needs of different CNN layers, benchmarks, or data sets. More specifically, the accelerator must be able to adapt to the image related parameters such as the input dimension, the neuron vector to be processed, and the hash size. For example, hash is critical for centroid calculation, which should be reconfigured in accordance with the benchmark in question, the number of images used, and the specific layer. Since CNN usually consists of a stack of layers, fast adaptation in hardware becomes even more critical. Therefore, number of activation maps in each layer can change and our design adapts according to these changes through reconfiguration.

Reconfiguration is not limited to the image related parameters, it also includes key generation (a dot-product operation) to be performed using optimal number of multipliers and adders. Furthermore, memory controller and memory units can be reconfigured accordingly.

5 Advanced Accelerator Design

5.1 Basic Architecture

We propose a tri-module custom architecture to perform the operations described in Section 4.1. High level architectural view of the proposed accelerator is given in Figure 5.

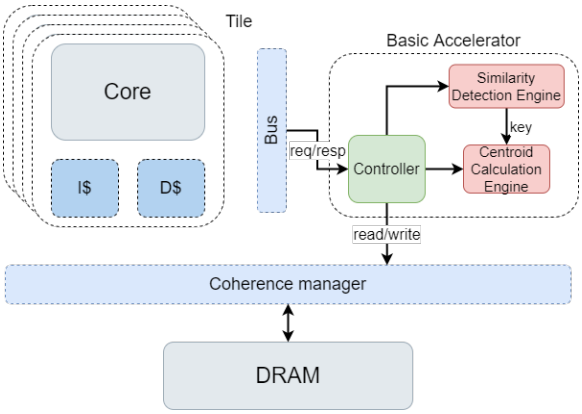


Fig. 5. Reuse-centric accelerator architecture.

Accelerator's main task is to group similar neuron vectors into the same cluster and calculate the centroid vector corresponding to these neurons. It performs the dot product of neuron vectors with the hash table entries to determine the clusterID. The resulting clusterID and centroid are stored in memory to be used later by the software or hardware platform utilizing the accelerator.

The overall tri-module architecture relies on three main components, namely *Similarity Detection Module (SDM)*, *Centroid Calculation Module (CCM)*, and *Main Controller (MC)*. The execution begins with requests done by CPU. MC issues signals required to control overall data flow between CPU, execution units, and the memory. In addition, it also ensures consistency and efficiency by preventing race conditions, replications, and unnecessary storage to memory. Based on the availability of CCM, which is responsible for keeping the clustered data up-to-date, SDM sends a signal to update centroid associated with the clusterID.

Similarity detection module calculates the clusterID for each input in a non-blocking pipelined fashion. If there are dependencies between neuron vectors, pipeline is stalled or data is forwarded to resolve these dependencies.

In our implementation, hardware/software platforms can control the accelerator through a communication interface, where a set of commands can be sent to the accelerator. For different kinds of network settings and memory requirements, users can generate the synthesizable RTL targeting both FPGA and ASIC. In addition, users can also reconfigure some network related parameters to customize the execution for different layers.

The details of each block are described in the following subsections.

5.1.1 Similarity Detection Module

Once the configuration settings are complete and hash tables are randomly initialized, input data start streaming into SDM. It completes its operation in L stages, in which, the first stage is dedicated to perform the dot product. Subsequent stages perform integer/floating point additions in a tree-like structure. Therefore, the total execution time of SDM consists of one multiplication (simultaneously done by K multipliers for all K neuron vectors) plus log_K number of additions. The resulting clusterID generated by the SDM is stored in an intermediate buffer. In case if multiple inputs with the same clusterID is received in the subsequent requests, pipeline gets stalled or the corresponding data is forwarded to ensure correct operation. Note that, in the base architecture implementation there are fixed number of multipliers and adders to perform these operations which puts a limit on the maximum number of neuron vectors supported. Obviously, the number of adders and multipliers can be incrased to support increased number of neuron vectors but this will result with inefficient and expensive implementation for small sized neuron vectors. As will be explained in the advanced architecture implementation, it is possible to adapt the design dynamically through reconfiguration.

5.1.2 Centroid Calculation Module

Centroid calculation unit mainly computes the centroid value for a cluster. It operates in five stages in a pipelined fashion as shown in Figure 6. These stages are detailed as follows:

- Stage 1 - Memory Load: It fetches previously calculated centroid and number of elements in that cluster from memory.
- Stage 2 - Multiply: Performs multiplication on previous centroid and element count.
- Stage 3 - Addition: Performs addition on new neuron vector and previous stage's output.
- Stage 4 - Division: Performs division on previous stage's output and new element count, generating the new centroid.
- Stage 5 - Write Back: Calculates the new centroid with new element count and places it into memory.

Fig. 6. Centroid Calculation Module.

5.1.3 Main Controller

Main Controller Unit (MC) coordinates requests/responses between CPU and accelerator. It also manages the data flow across computing units and memory. More specifically, it determines the availability of centroid calculation unit to ensure that a new input can be serviced in each clock cycle. The availability of centroid calculation unit depends on the generated clusterIDs and memory. While CCM is working on a specific input, if a new input with the same clusterID arrives for processing, it blocks and waits for the CCM to finish its operation to ensure memory coherency. To satisfy this requirement, a different input can be serviced during each clock cycle in a pipelined manner. More specifically, when the centroid is executing at the end of the fourth stage of CCM (i.e., Divison Stage), forwarding becomes possible for the incoming vector with the same clusterID.

5.1.4 Memory Interface

An open source bus interconnect called TileLink [15] is used to establish the connection between accelerator and memory, where a lightweight protocol without a cache is implemented. This protocol offers two types of operations to access the memory, namely, get and put. Moreover, basic primitives from available repositories is used to implement the memory interface. Similarly, communication with the off-chip DRAM is established by using available IPs from IP vendors. In addition, asynchronous queues are used to set up clock domain crossing between the DDR IP and the TileLink crossbar.

As one may expect, the main bottleneck in this accelerator setup is the time required to access the memory. This includes overheads in terms of both performance and energy. During memory read operation, pipeline needs to be stalled to avoid invalid transactions. These stalls slow down the accelerator, thus affecting the performance. Therefore, we focus on optimizing the requests from off-chip memory.

5.2 Advanced Architecture

As discussed earlier, two key problems in the basic architecture targeting reuse-centric approach are 1) dynamic change in the execution characteristics of images or layers, and 2) intensive memory requests. To address these two important issues, we extend our base architecture to include reconfiguration engine and memory support with custom caches. An overview of this advanced architecture implementation and internal components are given in Figure 7. We explain these advanced features in the following sections.

As can be seen from this figure, overall design consists of several stages. At first stage, global controller fetches the instructions from CPU host and writes neuron vectors into global buffer. Then, neuron vectors are fed into similarity detection engine (SDM), which runs in a non-blocking pipelined fashion. The key generated by SDM is used by global controller to update the valid array. Then, neuron vector and the associated key are sent to the id cache and cluster cache. Id cache keeps the key information for clustered neuron vectors while cluster cache updates the cluster centroid via moving average filter.

5.2.1 Reconfiguration Engine

Reconfigurability is considered as part of the design in order for the proposed accelerator to be used by different convolutional layers with different properties. For example, SDM has the capability of handling the clusterIDs for different vector sizes and for different number of hash functions. Therefore, it is critical to achieve reconfiguration in order to achieve efficiency and speed at the same time. Reconfiguration can be in two ways, either online or offline.

In the online reconfiguration, parameters of the accelerator are modified during a transition from one convolutional layer to another one. This way, it will be possible to use less resources such as adders, multipliers, or storage elements. However, note that, for a realistic execution scenario, there needs to be an upper limit in the supported convolutional layer properties since the accelerator will be implemented either on an FPGA or an ASIC. To capture this in our implementation, we set upper limits for various parameters such as hash size and input dimension. For instance, to run CIFAR10 on our accelerator, we need a maximum hash size of 15 and a maximum input dimension size of 10. With an online reconfiguration, these parameters can be reduced down to zero during some parts of the execution.

On the other hand, in offline reconfiguration, accelerator can be modified to be used for different networks based on the execution requirements. While offline reconfiguration is easier to manage, it provides less flexibility at runtime, which in turn may result with more energy consumption and under-utilization of the available resources. As explained earlier, layer by layer changes in a network will not be captured in such a scenario.

5.2.2 Memory Subsystem

There are different kinds of data that need to be accessed with different access patterns. For this reason, we extend our accelerator architecture to include different cache structures corresponding to each data type as shown in Figure 7. Details of these caches will further be explained in Section 5.2.4.

5.2.3 Memory Requirements

Reuse-centric implementation have multiple kinds of data associated with each convolutional layer and includes computation specific information. That is, each data object has different characteristics in terms of access pattern, size, and operations. Therefore, to achieve maximum efficiency in the advanced design, it is necessary to reflect on the different features of the memory subsystem architecture.

5.2.4 Memory Components

In this section, we describe memory subsystem architecture of the proposed accelerator. More specifically, in our memory subsystem, we manage each individual data separately by having independent caches for different kinds of data according to their access patterns as shown in Figure 7. In doing so, we 1) enable parallel accesses to multiple kinds of data and 2) optimize memory structure based on size of

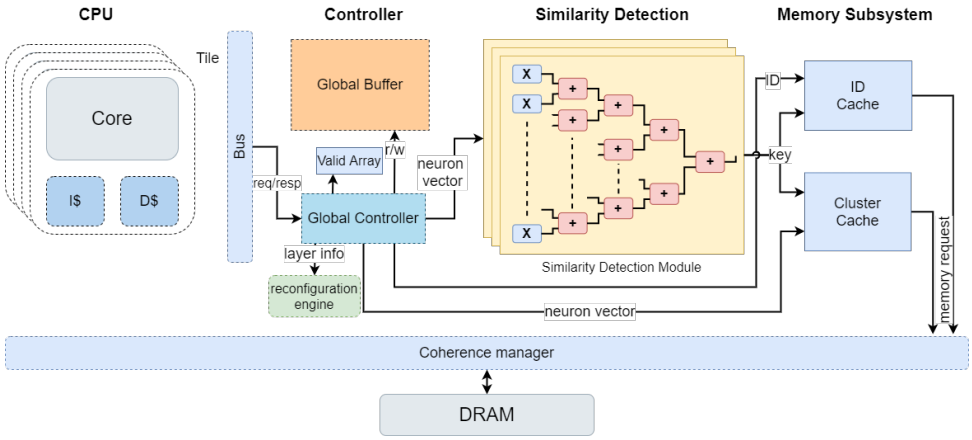


Fig. 7. Advanced accelerator design with reconfiguration, memory support and caches.

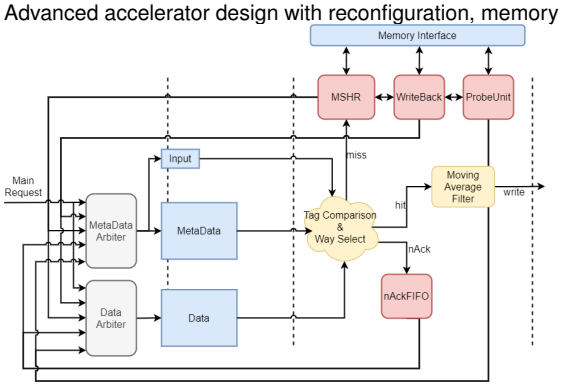


Fig. 8. Details of the caches used in the accelerator.

each data object and access characteristics. There are two main cache units and a global buffer used in the design:

- **Global Buffer**: This is an SRAM storage area where all the centroids for all the clusters are stored. This buffer is designed to eliminate the need for frequent memory requests. Details of this component will further be discussed.
- **Cluster Cache (CL$)**: Calculated centroid value for a cluster is stored in this cache. This cache is indexed with the ID of the cluster. Thus, count value indicating the number of elements in a cluster is also stored with the centroid. Data object size is directly proportional to the dimension of the layer and data size.
- **ID Cache (ID$)**: This cache is used to store the cluster ID corresponding to each input. Thus, data object size is related to the hash size.

Integration of the global buffer and caches with the rest of the advanced architecture is shown in Figure 7. In our current implementation, these caches are accessed in a non-blocking pipelined fashion. Detailed block diagram of these caches can be seen in Figure 8.

In the first cycle, it performs metadata and data read operations, followed by a tag comparison in the next stage. If tags are matched and access is granted, then related way is selected and response is sent. In case of a miss, miss status handling register (MSHR) is used to handle the memory request. MSHR ensures the coherency by checking current state of the cache entry. For instance, if it is dirty, MSHR flushes entire cache block via Write-back unit. When there is an index discrepancy between successive data requests, new requests are kept in a negative acknowledged FIFO, called nAckFIFO, until the problem is resolved. This is also when the FIFO is drained through the arbiters.

Moving average filter, which is found only in cluster cache, is used to calculate the new centroid on the fly. The formula we used for the calculation includes one addition, one subtraction and one division rather than one addition, one multiplication and one division. Eliminating the multiplication simplifies the implementation, even though it comes with some accuracy loss.

In order to extend our architecture for a multi-accelerator setup, we added a Probe Unit in both caches. If a probe request comes over the fully coherent bus interconnect, it establishes the cache coherency among different accelerator tiles.

In a cache implementation, it is important to keep temporal and spatial locality of each data object high for maximum throughput. These properties depend on the number of sets, number of ways, number of bytes in each block and number of words in a row. Therefore, various tests were performed for different combinations of these parameters to find the optimum set of values, as will be explained in Section 8.

5.2.5 Global Buffer

One of the main data structures required for our approach is the one holding the centroids of all clusters. Theoretically speaking, the maximum number of centroids equals to 2^H (H is the number of hashing functions). Multiplying it with the length of a subvector, L, we get $2^H * L$ as the maximum size of such a data structure. This size can be too large to always fit into SRAM or cache (Global Buffer in our case). However, in reality, the actual size of this data structure is much smaller due to its sparse nature. More specifically, the non-empty clusters weigh less than 13.3% of the maximal size in all our experiments.

Based on the insight, our design employs an adaptive scheme for implementing the data structure. When the maximum size $2^H * L$ is smaller than the buffer size (256KB in our implementation), the data structure takes the form of a directly accessible array. Otherwise, it takes an indirect sparse format, which consists of one indexing array CI, and one centroid array CA. When a new cluster is encountered,

its centroid is appended to the end of CA, and the index (a *short* integer) is assigned to $CI[i]$, where i is the numerical ID of that cluster (i.e., the corresponding H-bit vector from LSH). The space cost is $2^H * 2 + |CA|$.

As the worst-case storage requirement ($2^H * L$) for the respective layer of the benchmark can be easily computed, it is easy to enable direct or indirect access to the global buffer. The selection of the buffer size is given in more detail in the experimental setup.

5.2.6 Memory Request Handler

In case of a cache miss, MSHR is used to request data from memory. It consists of several states to keep the data coherent with the rest of the memory hierarchy.

State machine for MSHR starts with an invalid state. When there is a miss due to access permission, meaning that data exists in cache, tags are matched but requested operation is not allowed with the current access privileges, its metadata is updated by jumping to the meta-write state and then response is sent. Otherwise, there are two possible scenarios: If current permission is for read operation and requested permission is dirty, then data inside cache is written back to memory by changing to write-back states. If there is no need for write-back, meaning that data is not inside the cache, metadata is cleared and data is requested from memory through refill states. After data is taken from memory, metadata is updated and response is sent.

In addition, it is important to properly handle the outstanding DRAM access requests from all the internal memories and cache structures. Since our primary objective is to maximize throughput, we focus on increasing overall service rate of memory requests. In order to achieve this objective, overall memory bandwidth should be maximized. Memory bandwidth is determined by both on-chip memory bandwidth and off-chip memory bandwidth. Therefore, we need to allow parallel accesses to both on-chip memory and off-chip memory for maximum throughput. Our bus interconnect, TileLink, allows out-of-order completion for parallel requests to off-chip memory [15], which increases throughput considerably. Furthermore, we use different caches for different kinds of data and different MSHRs to control the memory accesses, thereby allowing parallel accesses to the on-chip memory as well, and improving the throughput further.

6 On-the-Fly Unfolding Module

The main purpose of this module is to support the efficient access to the input image. More specifically, using this module accelerator can avoid avoid unfolding the entire matrix in memory. As can be seen in Figure 9, unfolding module sits in between the global controller/buffer and Similarity Detection Module. This hardware module has an internal buffer to store part of an image, and provides neuron vectors to similarity detection module in order through the use of shift registers. Note that, this improves the efficiency drastically as same neuron vectors are repetitively used by different similarity detection. To reuse these shared neuron vectors, our unfolding module uses convolution window and moves data according to this window size. Both cluster cache data and similarity detection module gets updated with the coordination of global controller and similarity detection module. Note that, the size of the convolution and other parameters can be customized through the reconfiguration engine.

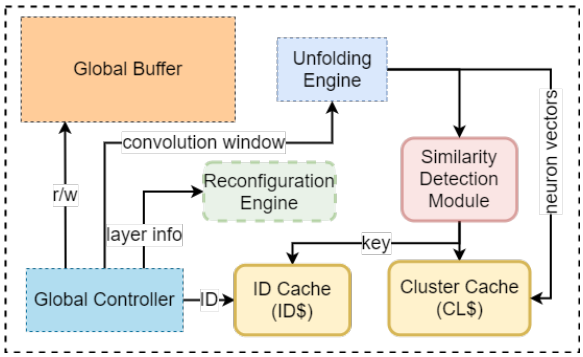


Fig. 9. Advanced architecture supporting on the fly unfolding operation.

7 Synergy with Other CNN Accelerators

While we integrate our accelerator into an existing software platform, it can also be integrated into other CNN accelerators and reduce the training and inference time as well. Many of the existing accelerators [1], [4], [16]–[18] are designed to accelerate the convolutional layer, which does the matrix matrix multiplication between the input and the weight matrices. Our accelerator module can serve as a preprocessing step and reduce the size of the input matrix being fed into the CNN accelerators. Therefore, an accelerator for the similarity detection has more profound impact and better re-usability than only being specific to a certain CNN implementation.

As an example, integration of the general reuse discovery engine into Eyeriss [19] is implemented. Eyeriss is implemented in hardware as a CNN dataflow using a spatial architecture with 168 processing elements. Our reuse discovery engine can be placed before the global buffer of Eyeriss as a pre-processing unit. Using similarity detection engine, centroids for similar input vectors are determined and stored inside local caches for later reuse. Finally, controller writes the centroids into the Eyeriss' global buffer to serve as input for systolic data setup. Our reuse discovery engine can be integrated into other CNN accelerators similarly.

8 Evaluation

This section reports the performance improvement brought by the reuse-centric accelerator (basic and advanced designs) to both software implementations and other CNN accelerators. We also report the energy cost and a sensitivity study on the architectural design parameters.

8.1 Experimental Setup

We have implemented our base and advanced accelerator using Xilinx's Vivado [20] tool on VCU118 [21] evaluation board and collected the energy and resource utilization. We measured performance using verilator [22] on the RTL generated from the design written in Chisel [23]. In addition,

we used DRAMSim2 [24] tool to model a DDR3 DRAM. Furthermore, we used Scale-Sim [25], [26] tool to measure the benefits on performance when our accelerator is integrated into Eyeriss [19]. Note that, while we implement the accelerator for general purpose CPUs, it can also be used with GPUs. Since it is an orthogonal technique, our proposed architecture can be embedded into GPUs to further reduce the overhead of similarity detection among neuron vectors.

8.1.1 Benchmarks

We use four convolutional neural networks to evaluate our accelerator: CifarNet [27], AlexNet [28], VGG-19 [29], and MobileNet [30]. Table 3 lists datasets running on described networks.

TABLE 3
Convolutional network and dataset pairs used in evaluation.

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

8.1.2 Datasets

We use Cifar10 [27] and Imagenet [31] as our datasets for collecting the experimental results. Cifar10 has 60000 images with a size of 32x32, whereas Imagenet dataset has more than 14 million images with a size of 224x224.

8.1.3 Default Parameters

Selection of the optimal H and L parameters for a specific network is achieved through hyper-parameter optimization during training, as described in [9]. It is important to note that lower values for H and L are desired, provided that there is no accuracy loss during the inference. For our benchmarks, the minimum and maximum values for H and L are, $H = \{10, 15\}, L = \{5, 10\}$ for CifarNet, $H = \{15, 16\}, L = \{11, 24\}$ for AlexNet, $H = \{12, 20\}, L = \{9, 18\}$ for VGG, and $H = \{8, 18\}, L = \{3, 16\}$ for MobileNet, respectively.

Our default accelerator frequency is 50MHz and the global buffer size is 256KB. As explained in the advanced architecture details, we use a global buffer to hold the centroids of all clusters. According to the runtime characteristics, the largest space required is 206KB by the second layer of VGG. In order to be able to satisfy the requests from within the accelerator, we set the size of this buffer as 256KB. Most of the layers of our benchmarks fit within this space directly. More specifically, for all the layers of CifarNet, 4 layers of VGG, and 10 layers of MobileNet, centroids directly fit into the global buffer. On the other hand, all the remaining layers fit using indirect access mechanism.

We performed a design space exploration to understand the effects of caches with the aggregated workloads as shown in Figure 10. Based on these results, we set the default cache parameters as in Table 4.

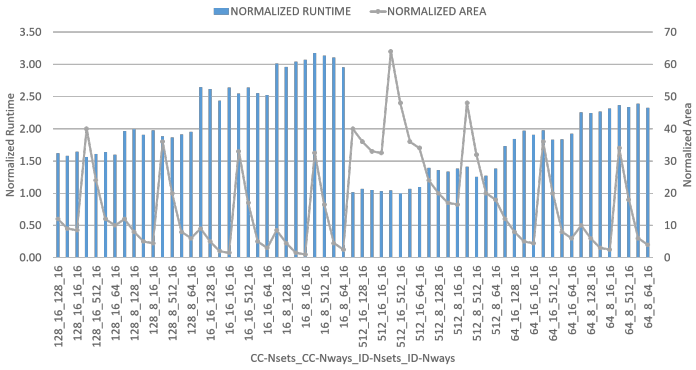


Fig. 10. Design space exploration for the advanced design.

TABLE 4
Selected parameters used for caches.

Parameters	Cluster Cache	ID Cache
nSets	512	16
nWays	8	16
nBBs	64	64
rowBits	256	256
nMSHRs	4	4

8.1.4 Execution environments

We have four execution environments whose details are as follows:

1) **CPU:** A software based implementation running on a mobile CPU is compared against our accelerator. The specific embedded system used in our experiments is a Huawei SE Mate mobile phone, which has a Huawei HiSilicon KIRIN 659 processor with 8 cores (4×2.36 GHz Cortex-A53 + 4×1.7GHz Cortex-A53), and 64GB internal storage.
2) **Basic Accelerator:** Our basic reuse-centric architecture as explained in Section 4.
3) **Advanced Accelerator:** Our advanced reuse-centric architecture as detailed in Section 5.
4) **Eyeriss + Advanced Accelerator:** We further integrate our accelerator into an existing state of the art accelerator Eyeriss [19] in order to observe potential benefits in runtime. We use SCALE-sim [25], [26], a CNN accelerator simulator, to study the performance in terms of run time cycles.

8.2 Performance and Synergy with Existing Accelerators

Previous studies [9] on software methods have already reported that reuse-based convolutions achieve almost identical inference accuracies as the original CNNs do. The hardware accelerator does not change the accuracy. We hence focus the discussion on performance.

We evaluate the performance of our accelerator for the aforementioned networks and compare them with the mobile platform. We first study the performance of the similarity discovery module. As shown in Figure 11, the basic design accelerates the similarity discovery by up to 5.5X, compared to software implementations. The advanced design further enhanced the performance and gives up to 14.96X speedups. The advanced design performs much better than the basic design for all cases due to the memory support provided by the custom caches. Comparing the

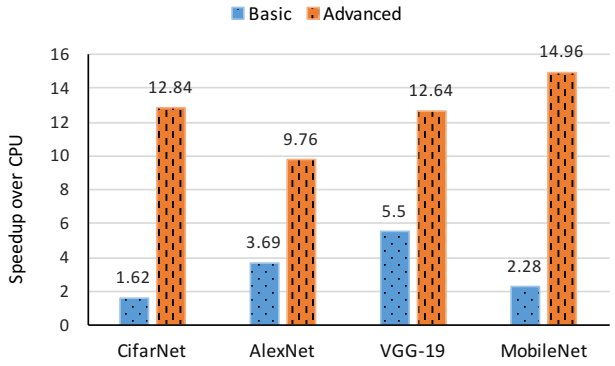


Fig. 11. Average speedups over mobile CPU obtained through basic and advanced design for the similarity discovery module.

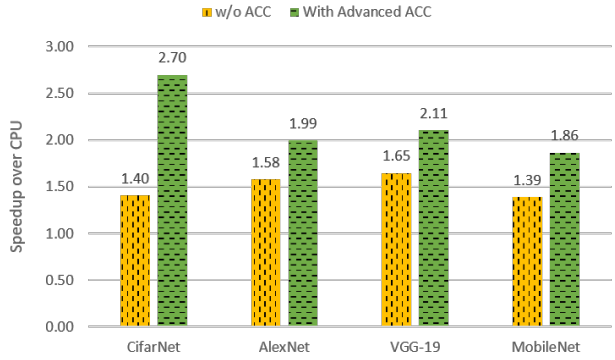


Fig. 12. Speedup for reuse-centric implementations in pure software and with the proposed accelerator. The baseline is the default convolution on mobile CPU.

more complex network architectures (AlexNet, VGG-19 and MobileNet), we can see that as the network goes deeper, the advanced design achieves larger speedup. In some cases, the performance improvements translate into relatively modest speedups when the entire execution is considered as shown in Figure 12. As expected, convolutional layers' execution time mostly dominates the other layers. However, the reason for not fully utilizing the accelerator is due to the fact that the other parts of the convolution (unfolding, full result derivation, and centroid matrix multiplication) are implemented in software.

When the other parts are also accelerated, the benefits from the reuse-centric accelerator becomes much more significant. For instance, Eyeriss accelerates the 4th convolutional layer of AlexNet by 85.9X speedup [19] over the mobile CPU. Our experiments show that after reuse-centric accelerator is integrated into Eyeriss, Eyeriss runs 1.08X faster on average on AlexNet, as it processes only the centroid matrix-multiplication, which is much smaller than the original. That translates to 92.8X speedups over mobile CPU.

It is important to note that the global data reuse technique is compatible with all existing sparsity techniques. The granularity of sparsity can be categorized under four groups: fine-grained sparsity (zero-dimensional), vector-level sparsity (one dimensional), kernel-level sparsity (two dimensional), filter-level sparsity (three dimensional), from irregular to more regular [32]. It speeds up the similarity detection process, especially for higher-level sparsity techniques, since we process one vector at a time. This way,

TABLE 5
Resource utilization of the advanced accelerator on VCU118 board.

Module Name	Total LUTs	Logic LUTs	LUT RAMs	FFs	RAM B36	RAM B18
broadcast	870	870	0	356	0	0
fetchBuffer	5464	25256	208	13	0	0
lsh	27399	27147	252	19281	158	16
CL$	5353	5231	122	1774	56	16
ID$	4285	4155	130	1286	70	0
indexModule	17489	17489	0	15066	0	0
memVal	55	55	0	2	32	0
ddr_infra	39	39	0	65	0	0
ddr_ram	12261	11394	816	15907	25	1

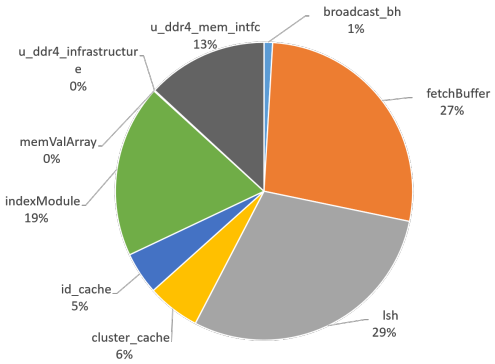


Fig. 13. Area breakdown of the modules in the advanced architecture.

it reduces the overall runtime. Besides, even with fine-grained sparsity, skipping data locations marked as zero lowers energy consumption. As a result, all levels of sparsity techniques will benefit the similarity discovery engine.

8.3 Area Breakdown and Resource Utilization

For the configurations given in Table 4, our accelerator is synthesized and implemented through Xilinx's Vivado tool. Area breakdown of the accelerator based on the number lookup tables (LUTs) used during implementation on FPGA is shown in Figure 13. Main module, LSH and the global controller occupies the largest area. As can be seen from this figure, caches take up a small space compared to others. Resource utilization of the main modules in our advanced design is given in Table 5.

9 Related Work

There are a bunch of work on hardware DNN accelerators. Among them, FPGA [1], [4], [33]–[37] is well studied and shows great potential on accelerating DNN inference. [33] proposed a memory-centric design flow to synthesize reconfigurable accelerator template for CNN. It maximizes the on-chip memory usage with optimized data access patterns. With hand-optimized design templates, DNNWEAVR [1] can automatically generates synthesizable accelerators for given DNN-FPGA pairs. It translates a high-level specification in Caffe to its novel macro dataflow ISA and provides a heuristic algorithm to tile, schedule, and batch DNN operations in order to minimize off-chip memory access, maximize data reuses and the utilization of the FPGA's memory and other resources. Alwani and other researchers [37] propose a new CNN evaluation strategy on FPGA,

which modifying the order in which the original input data are brought on chip. A new design methodology [4] has been proposed to solve the inefficiency issue caused by reconfiguring FPGA for CNN layers with different dimensions. While all these efforts focus solving the external memory access problem and efficient reconfigurability problem, they all try to implement the whole layer on CNN and run the whole CNN network on FPGAs. Our work tries to accelerate only the LSH related computation introduced by *reuse-centric CNN acceleration*. While we also face the memory bandwidth problem and the reconfigurability problem, we are forced to explore other ways to solve the problems because of the intrinsic characteristics of *reuse-centric CNN* as discussed in Section 2.

Some recent work [2], [5], [6] proposes designs of hardware accelerators to explore computation reuses in CNN. Riera et al. [2] points out that a large number of inputs of the fully-connected and convolutional layers have the same quantized value as in the previous DNN execution for sequence processing problems such as speech recognition, video classification and self-driving cars. They propose a computation reuse scheme based on this input similarity and implements this reuse-based inference technique on top of a state-of-the-art DNN accelerator. Similarly, EVA [6], as an extension to state-of-the-art CNN accelerator design, is a hardware module to detect the changes in the visual input and incrementally update a previously-computed activation. It implements an approximately incremental CNN execution approach and adaptively controls the trade-off between inference accuracy and resource efficiency. An other work [5] proposes UCNN, an accelerator to exploit weight repetition in and across CNN filters. It reuses CNN subcomputations and reduces CNN model size. The first two work [2], [6] only explores temporal reuse opportunities on inputs while UCNN [5] only explores the spatial reuse opportunities on weight. Comparing to them, our general reuse-centric CNN accelerator exploits both the temporal and spatial reuse opportunities on the inputs. The similarities it explores are much more general than the aforementioned accelerators.

Researchers also propose hardware solutions for accelerating the LSH based algorithms [38]–[41]. Ternary Locality Sensitive hashing [38] is proposed to solve the LSH based nearest neighbor problem using ternary content addressable memory. It achieves a space requirement almost linear to the data base size and $O(1)$ query time. Jiang and Maya [39] propose using FPGA to accelerate the k-Nearest Neighbors (k-NN) algorithm. They adopt the LSH for approximate k-NN and optimize the main processing component in FPGA for this LSH-based algorithm to achieve high throughputs. A hierarchy staged LSH (HSLSH) [40] takes advantages of the hierarchical memory structure (the fast and large memories) and realizes accelerations on Audio Fingerprint Search. These works all try to accelerate algorithms which has LSH as the core computation. However, none of them take a deep dive into the design of LSH on FPGA nor consider optimizing the architecture for LSH. Our work implements a hardware accelerator, which reduces the overhead of neuron vector similarity detection and reduces the energy consumption of reuse-centric CNN inference. It includes a custom memory subsystem with different cache structures along with reconfigurability features.

10 Conclusion

In this paper, we introduce the concept of *general reuse*, and propose a hardware solution for the *reuse-centric CNN acceleration*, where a customizable accelerator architecture is implemented to reduce the energy consumption and to boost the performance compared to an embedded platform. We synthesized and implemented our design and tested with four CNN networks and two popular datasets. Except small layers present in the benchmarks, our accelerator outperforms the mobile platform both in terns of energy and performance. Specifically, we see up to 14.96X speedups for similarity discovery, up to 3.33X speedups for a convolutional layer. We expect these results to be an order of magnitude better when implemented on an ASIC.

11 Acknowledgment

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

References

[1] 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.

[2] 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*, ser. ISCA '18. Piscataway, NJ, USA: IEEE Press, 2018, pp. 57–68.

[3] 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*, ser. ISCA '18. Piscataway, NJ, USA: IEEE Press, 2018, pp. 674–687.

[4] 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.

[5] 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.

[6] 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.

[7] 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: MIT Press, 2015, pp. 1225–1233.

[8] 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: ACM, 2004, pp. 253–262.

[9] 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*, ser. ICS '19, 2019.

[10] J. J. Dongarra, J. Du Croz, S. Hammarling, and I. S. Duff, "A set of level 3 basic linear algebra subprograms," *ACM Trans. Math. Softw.*, vol. 16, no. 1, p. 1–17, Mar. 1990.

[11] 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.

[12] 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*, 1998, pp. 604–613.

[13] 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*, 2006, pp. 459–468.

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

[15] H. Cook, W. Terpstra, and Y. Lee, "Diplomatic design patterns: A tilelink case study," in *Proceedings of First Workshop on Computer Architecture Research with RISC-V*, Boston, MA, USA, 2017.

[16] 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.

[17] 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 *Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on*, 2016.

[18] 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. [Online]. Available: http://arxiv.org/abs/1807.07928

[19] 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*, 2016, pp. 262–263.

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

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

[22] "Veripool." [Online]. Available: https://www.veripool.org/

[23] t. C. Developers, "Chisel/firrtl: Home." [Online]. Available: https://www.chisel-lang.org/

[24] P. Rosenfeld, E. Cooper-Balis, and B. Jacob, "Dramsim2: A cycle accurate memory system simulator," *IEEE Comput. Archit. Lett.*, vol. 10, no. 1, p. 16–19, Jan. 2011. [Online]. Available: https://doi.org/10.1109/L-CA.2011.4

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

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

[27] A. Krizhevsky, "Learning multiple layers of features from tiny images," Tech. Rep., 2009.

[28] 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: Curran Associates Inc., 2012, pp. 1097–1105.

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

[30] 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. [Online]. Available: http://arxiv.org/abs/1704.04861

[31] 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.

[32] H. Mao, S. Han, J. Pool, W. Li, X. Liu, Y. Wang, and W. J. Dally, "Exploring the granularity of sparsity in convolutional neural networks," in *2017 IEEE Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, 2017, pp. 1927–1934.

[33] M. Peemen, A. A. A. Setio, B. Mesman, and H. Corporaal, "Memory-centric accelerator design for convolutional neural networks," in *2013 IEEE 31st International Conference on Computer Design (ICCD)*, 2013.

[34] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing fpga-based accelerator design for deep convolutional neural networks," in *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2015.

[35] X. Han, D. Zhou, S. Wang, and S. Kimura, "CNN-MERP: an fpga-based memory-efficient reconfigurable processor for forward and backward propagation of convolutional neural networks," *Proceedings of the 34th IEEE International Conference on Computer Design, ICCD 2016*, 2016.

[36] H. Li, X. Fan, L. Jiao, W. Cao, X. Zhou, and L. Wang, "A high performance fpga-based accelerator for large-scale convolutional neural networks," in *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, 2016.

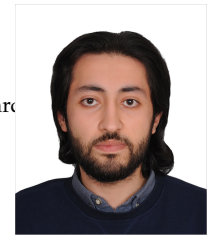
[37] M. Alwani, H. Chen, M. Ferdman, and P. Milder, "Fused-layer cnn accelerators," in *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, 2016.

[38] R. Shinde, A. Goel, P. Gupta, and D. Dutta, "Similarity search and locality sensitive hashing using ternary content addressable memories," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, 2010.

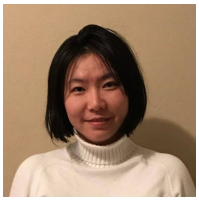
[39] W. Jiang and M. Gokhale, "Real-time classification of multimedia traffic using fpga," in *2010 International Conference on Field Programmable Logic and Applications*, 2010.

[40] M. Fukuda and Y. Inoguchi, "Probabilistic strategies based on staged lsh for speedup of audio fingerprint searching with ten million scale database," in *Proceedings of the 8th International Symposium on Highly Efficient Accelerators and Reconfigurable Technologies*, 2017.

[41] V. T. Lee, J. Kotalik, C. C. d. Mundo, A. Alaghi, L. Ceze, and M. Oskin, "Similarity search on automata processors," in *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2017.



Nihat Mert Cicek obtained his bachelor degree from electrical engineering and computer engineering (double major) department of Istanbul Technical University. Currently, he is an MSc student in the Department of Computer Engineering at Bilkent University and is a full-time employee at Aselsan Inc. His research interests include computer architecture, digital VLSI, AI, DL, memory and memory subsystem.



Lin Ning received her Bachelor's degree and Master's degree in Physics from Huazhong University of Science and Technology and North Carolina State University respectively. In 2015, she started to work with Dr. Xipeng Shen as a Ph.D. student in Computer Science at North Carolina State University. Her research interest resides in the high-performance Machine Learning and Deep Learning.

Ozcan Ozturk is a Professor in the Department of Computer Engineering at Bilkent University. His research interests are in the areas of manycore architectures, parallel computing, and computer architecture. Prior to joining Bilkent, he worked at Intel and Marvell. He also held positions in NEC Labs, North Carolina State University and Arizona State University. His research has been recognized by Fulbright, Turk Telekom, IBM, Intel, HiPEAC, Tubitak, and European Commission.

Xipeng Shen received the PhD degree in computer science from the University of Rochester, in 2006. He is a professor in Computer Science at the North Carolina State University.He is a receipient of the DOE Early Career Award, NSF CAREER Award, Google Faculty Research Award, and IBM CAS Faculty fellow Award. He is an ACM distinguished member, ACM distinguished speaker, and a senior member of the IEEE. His interest is in Programming Systems and Machine Learning.