GCONV Chain: Optimizing the Whole-life Cost in End-to-end CNN Acceleration

Jiaqi Zhang, *Student Member, IEEE*, Xiangru Chen, *Student Member, IEEE*, and Sandip Ray, *Senior Member, IEEE*

Abstract—The acceleration of CNNs has gained increasing attention since their success in computer vision. Since the heterogeneous layers cannot be processed by accelerators proposed for convolution layers only, modern end-to-end CNN acceleration solutions either transform diverse computation into matrix/vector arithmetic, which loses data reuse opportunities in convolution, or introduce dedicated functional unit to each kind of layer, which results in underutilization and high update expense. To enhance the whole-life cost efficiency, we need a solution that is efficient in processing CNN layers and has the generality to apply to all kinds of existing and emerging layers. To this end, we propose GCONV Chain, a method to convert the entire CNN computation into a chain of standard general convolutions (GCONV) that can be efficiently processed by existing CNN accelerators with low-overhead hardware support. This paper comprehensively analyzes the GCONV Chain model and proposes a full-stack implementation to support GCONV Chain. Our results on various CNNs demonstrate that GCONV Chain improves the performance and energy efficiency of existing CNN accelerators by an average of 3.4x and 3.2x respectively. Furthermore, we show that GCONV Chain provides low whole-life costs for CNN acceleration, including both developer efforts and total cost of ownership.

Index Terms— Computer architecture, convolution neural network, hardware acceleration, neural network.

1 Introduction

Since its resurgence, Convolutional Neural Network (CNN) has demonstrated impressive success in promoting the computer vision in a wide range of applications [1]–[3]. However, the high accuracy of CNN is achieved at the cost of enormous computation and data movement, which is an undesirable obstacle to widely implementing and deploying them. Consequently, CNN acceleration has received increasing attentions.

Normally, CNN computation and parameters are dominated by the convolution layers. Based on this fact, abundant prior works [4]–[12] focus on the acceleration of these layers by designing customized architectures and dataflows to enhance the performance and data reuse in convolution operations (we classify these accelerators as **CIP**, i.e., convolution intended processors). However, recent CNNs incorporate more heterogeneous functional layers. For example, Fig. 1(a) depicts a basic block of MobileNet [13] with four various layers. Except for the first layer, each of them performs unique computation that cannot fit into the traditional definition of a convolution layer and thus cannot be accelerated by CIPs as illustrated in Fig. 1(b). Since these non-traditional layers play an important role in promoting the accuracy [14] of CNNs and are even proved to have better learning capability than the traditional layers [15], overlooking them can lead to degraded accuracy. Therefore, CIPs that only accelerate the convolution layers and are even incapable of parsing the other layers have to offload them to somewhere else, failing to efficiently perform the end-to-end CNN computation.

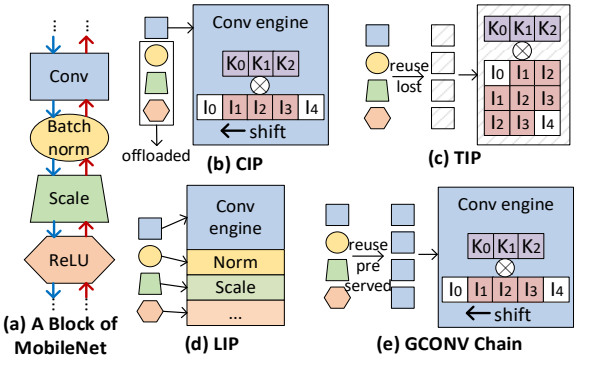

Fig. 1. Three Types of Modern CNN Accelerators and GCONV Chain

To deal with the "elephant in the room", a common practice is to transform the computation in diverse functional layers into tensor (mostly matrix and vector) operations and perform them in the tensor instruction processors (**TIP**) [16]–[19]. As illustrated in Fig. 1(c), TIPs are able to process any CNN layer but they lose certain data reuse opportunities compared with CIPs and thus result in low data movement efficiency. TIPs also suffer from lower code density because they explicitly manage the data loading and each matrix/vector operation. Another plausible solution is to add a dedicated on-chip processing unit for each type of layer in addition to the convolution (these accelerators are categorized as **LIP**, i.e., the layer instruction processors) [20]–[22], as illustrated in Fig. 1(d). Nevertheless, almost every new network features a new functional layer. It is costly for the hardware developers to design a new component to process the newly introduced layer for the full-fledged accelerators and for the users to upgrade their deployment. What is more, the varying CNN structures make it impossible to design a

• *Jiaqi Zhang, Xiangru Chen, and Sandip Ray are with the Department of Electrical and Computer Engineering, University of Florida, Gainesville, FL 32611. E-mail: jiaqizhang, cxr1994816@ufl.edu, and sandip@ece.ufl.edu.*

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2021.3128159, IEEE Transactions on Computers

heterogeneous accelerator with high utilization for all the workloads.

To understand the issues involved, it is helpful to use analogy from instruction set architecture. In particular, think of TIPs as analogous to RISC and CIPs and LIPs to CISC. The challenges faced by the existing works drive us to find *a balance point between the generality of RISC-like and efficiency of CISC-like CNN accelerators*. We innovatively propose to convert the end-to-end diverse CNN computation into a series of general convolution (**GCONV Chain**), where the operations are as uniform as that in TIPs and the reuse opportunities can be leveraged as in CIPs and LIPs, as shown in Fig. 1(e). Specifically, though the diverse layers perform different functions, we develop a parameterized model that expresses all the CNN operations as a scale-up of a simple 1-D GCONV. For example, the local response normalization [23] and batch normalization [14] can be viewed as general convolutions in the channel dimension and batch dimension respectively. This enables the end-to-end CNN to be efficiently accelerated by a uniform computation fabric, where the complex acceleration problem is simplified to a single mapping algorithm of the 1-D GCONV.

The benefits of our approach are threefold: (1) since GCONV Chain is composed of general convolutions and CNN accelerators are designed to perform convolution, it can be applied to almost any CNN accelerator with no worse if not better performance (Section 6.3); (2) when GCONV Chain is applied to CIPs, it eliminates the costly offloading overhead. And this GCONV armed CIP (GC-CIP) does not suffer from the low data movement efficiency in TIPs and resource underutilization in LIPs, showing that GC-CIPs are promising accelerators for the entire end-to-end diverse CNN computation (Section 6.5); (3) the generality of GCONV Chain further makes GC-CIPs the most whole-life cost efficient choice by lessening the burden of both users and developers (Section 6.6).

The paper makes the following contributions:

(1) We recognize the gap between modern CNNs composed of diverse functional layers and the existing CNN accelerators which are proposed for either generality or efficiency for certain layers.

(2) We propose GCONV Chain, a method to convert the end-to-end CNN computation into a chain of general convolutions (GCONVs) so that all the CNN layers can be efficiently and uniformly accelerated.

(3) We further propose a full-stack GCONV Chain implementation on existing CNN accelerators, including GCONV Chain generation, acceleration and hardware support.

(4) Our experiments on seven CNNs from a wide range and five representative accelerators of different types show that GCONV Chain can significantly improve the performance and energy efficiency of existing CNN accelerators and optimize the cost of both development and ownership.

The rest of this paper is organized as follows. Section 2 provides the background on CNN acceleration and the motivation for CNN generalization. Section 3 describes the approach to transform CNN computation into a GCONV Chain. Then Section 4 introduces the methodologies to analyze the GCONV workload and the algorithm to accelerate GCONV Chain. Section 5 elaborates the system implementation of GCONV Chain. Sections 6 evaluates our proposed GCONV Chain. And Sections 7 and 8 discusses the related work and concludes the paper respectively.

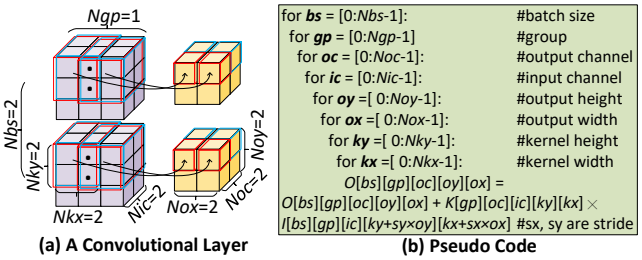

Fig. 2. A Traditional Convolution Layer

2 Background and Motivation

2.1 A Traditional Convolution Layer

CNNs are neural networks featured by the dominating convolution layers. Fig. 2(a) and (b) provide the illustration and pseudo code of a traditional convolution layer to help understand the computation pattern and reuse opportunities in the convolutional operation, where *Np* stands for the argument of parameter *p*. The same set of *Noc* kernels are applied to a batch of *Nbs* input samples. The size of each kernel is *Nky*×*Nkx*×*Nic* and they slide on the height and width of the input channels to generate *Noc* channels of *Noy*×*Nox* outputs. In some CNNs [23][13], the input channels are divided into *Ngp* groups to perform convolution individually.

There are two kinds of reuses in convolution, *parallel-reuse* and *overlap-reuse*. Parallel-reuse results from parallel computation in terms of a certain data. For example, each kernel is reused to generate outputs within each channel and the inputs are reused by different output channels. Similarly, each output is reused within the kernel since the partial results are reduced to generate one output. The sliding of the kernel also brings overlap-reuse when the input windows of consecutive outputs are overlapped. For instance, in Fig. 2(a), the inputs with dots are shared by the computation of different outputs, multiplied by different weights. Exploiting these two kinds of parallelism in the convolutional operation has been widely recognized as an effective method to reduce the costly data movement in CNN acceleration.

2.2 Non-Traditional Layers in CNNs

Aside from the convolution layers, there is a variety of other necessary layers in the CNNs, each of which performs a different operation on the input data. And the operation of a same layer even varies in the training and inference of the network. Besides the traditional layers such as fully connection, max pooling, ReLU and softmax that breed the success of the nascent CNN model, i.e., LeNet [24], the growing power of the emerging CNNs is always accompanied with new functional layers. Benchmarked on seven popular CNNs in the timeline (i.e., four classification CNNs: AlexNet (AN) [23], GoogLeNet (GLN) [25],

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2021.3128159, IEEE Transactions on Computers

TABLE 1
CNN and Accelerator Characterization

(a) Non-Traditional Layers in CNNs							(b) Inefficiencies of Accelerators		
CNN type	Net-work	New layer types	Non-traditional ratio				Data replication (TIP)	Offloading (CIP)	Utilization (LIP)
			Layers	Computation	Data footprint	Movement			
classifi-cation	AN	LRN, dropout	24%	<1%	5%	<1%	35x	3%	98%
	GLN	ave pool, concat	13%	<1%	17%	1%	6x	13%	80%
	DN	batch norm, scale	66%	5%	76%	22%	2x	53%	17%
	MN	depthwise conv	62%	8%	73%	16%	2x	77%	11%
other	ZFFR	RoI, proposal	29%	<1%	41%	<1%	4x	57%	86%
	C3D	3D conv, 3D pool	52%	99%	46%	99%	6x	87%	1%
	CapNN	prim, digicaps	18%	95%	6%	93%	3x	33%	1%

DenseNet (DN) [26], MobileNet (MN) [13], in addition to an object detection network Faster R-CNN+ZFNet (ZFFR) [27][28], a 3-D CNN for video processing (C3D) [29] and a capsule neural network (CapNN) [30]), Table 1(a) lists the newly introduced layers and the ratio of these non-traditional layers. As can be observed in Columns 5 to 7, the non-traditional layers account for increasing computation, data footprint and movement in the CNNs. And these layers are taking significant roles in determining the speedup and energy efficiency of CNNs.

2.3 Modern CNN Acceleration Challenges

As CNNs are being employed more frequently and the heterogeneity in CNN layers keeps increasing, it is obviously important to have a solution that efficiently accelerates all the CNN computation. Unfortunately, we notice that almost all the existing works suffer from various inefficiencies when processing modern CNNs.

TIP: The tensor instruction processors (TIP) are able to process any CNN layer by transforming them into tensor (mostly matrix and vector) operations [16]–[19]. However, they cannot exploit the abundant overlap-reuses in CNNs. For instance, *im2col* [17] is commonly employed by TIPs to transform convolution into matrix multiplication, where the input windows are flattened to columns in a matrix and then multiplied by a weight matrix, as shown in Fig. 1(c). This results in the replication of the input data (marked in red) and thus low energy efficiency. Column 1 in Table 1(b) quantifies the total data replication of the CNNs in TIPs. Since the power consumption is dominated by the data movement [19], this overhead significantly increases the operating expense for the users.

CIP: The main component of a convolution intended processor (CIP) is a convolution engine, which implements various exhaustively explored dataflows (e.g., weight stationary, output stationary, row stationary, etc.) to maximally exploit both the parallel and overlap-reuse opportunities in convolution [4]–[12]. For example, the convolution engine in Fig. 1(b) adopts the weight stationary dataflow, where the inputs are shifted along the stationary weights to exploit the overlap-reuse, avoiding data replication. However, since the proposed dataflows only apply to the computation model of traditional convolution layers, CIPs are inefficient or even incapable of parsing the parameters when processing the other layers. Therefore, offloading is required for non-traditional layers whose acceleration is omitted in CIPs. Column 2 in Table 1(b) characterizes the ratio of intermediate data that requires offloading for a series of non-traditional processing. Note that since the offloading energy consumption can be as high as 146x of the on-chip data movement in our experiment, this adds considerable burden to the system.

LIP: The layer instruction processors (LIP) [20]–[22] add a dedicated on-chip processing unit for each type of layer in addition to the convolution engine and process the corresponding layers of different inputs in a pipeline. Resulted from the variation of the number and shape of each kind of layers in a certain CNN, pipeline bubbles are unavoidable. To implicate this, Column 3 in Table 1(b) lists the utilization of different networks assuming the pipeline has two stages for traditional and non-traditional layers respectively. The resources are partitioned based on the ratio of the traditional and non-traditional computation in all the networks. As observed, the uniform partitioning results in significantly varying utilization. And the utilization is extremely low in networks with more non-traditional computation (e.g., C3D and CapsuleNN). Layers that pose a pipeline barrier also lower the utilization (e.g., batch normalization in DenseNet and MobileNet). What's more, LIPs demand the developers to design an efficient acceleration solution to any new layer and the users to update the devices as frequently as the evolution of CNNs.

To enhance the whole-life cost efficiency of end-to-end CNN computation, we need an acceleration solution that is efficient in processing CNN layers and has the generality to apply to all kinds of existing and emerging layers.

3 General Convolution (GCONV) Chain

Our goal is to convert the diverse CNN layers into a chain of standard operations without losing the convolution pattern. Instead of breaking them down into basic matrix/vector arithmetic as in TIPs, we view them as multi-dimension convolutions under a more generalized definition. This section proposes the GCONV model and the method to transform the entire CNN computation into a GCONV Chain.

3.1 GCONV Operation Definition

GCONV is the most basic operation in our system. It is a concisely parameterized 1-D convolution which can be scaled up to multiple dimensions to define various CNN operations. Compared with the traditional definition of a convolution layer, the *simplicity*, *scalability* and *representability* of GCONV as discussed below make it ideal to

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2021.3128159, IEEE Transactions on Computers

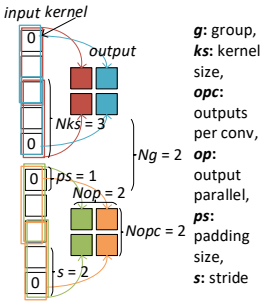

Fig. 3. 1-D GCONV

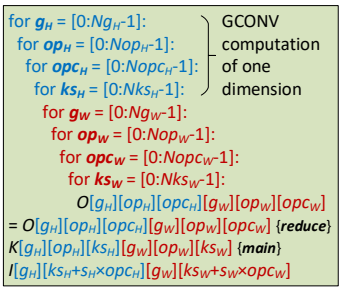

Fig. 4. Pseudo Code of 2-D GCONV in *H* and *W* Dimensions

effortlessly, uniformly and efficiently accelerate all kinds of CNNs.

Definition: Fig. 3 shows an example of 1-D GCONV. It is characterized by four parameters that form the nested loop of the computation as shown in Fig. 4: the inputs are separated into ***Ng*** groups and no connection or reuse exists inter-group; in each group, the inputs are convolved by ***Nop*** kernels in parallel; each kernel has ***Nks*** weights; ***Nopc*** outputs are generated by each kernel. There are also two auxiliary parameters, i.e., ***ps*** for padding size and ***s*** for stride, as in the traditional convolution. The input size *Nipc* of 1-D GCONV is not required since it can be derived as:

$$Nipc = (Nopc + 1) \times s + Nks - 2ps \quad (1).$$

Note that dilation is not modeled in GCONV since we adopt the technique proposed in [31] to turn the dilated convolution into dense convolution by simply reordering the computation.

Simplicity: Recall that the traditional convolution layer in Fig. 2 has eight different parameters. The 1-D GCONV is relatively simple with only four. These four parameters are necessary to preserve the data reuse patterns. Specifically, the input, kernel and output can be parallel-reused for *Nop*, *Nopc*, *Nks* times and the input can be overlap-reused when $Nks > s$. In our system, if not explicitly denoted, the parameters take default values as [*ps*: 0, *s*: 1, *Ng*: 1, *Nop*: 1, *Nks*: 1, *Nopc*: 1].

Scalability: This simple 1-D GCONV can easily scale up to multiple dimensions. The dimension of GCONV is determined by the data in the network. For example, the convolution layer in Fig. 2 manifests four dimensions: the mini-batch (***B***), channel (***C***), height (***H***) and width (**W**). Fig. 5 shows the view of Fig. 2(b) from each dimension, where the traditional convolution parameters are replaced (marked with strikethrough) by the GCONV parameters. While the operations in dimensions *H* and *W* are naturally 1-D convolution, it might be counter-intuitive that the operations in *B* and *C* can also be described as 1-D GCONVs. Specifically, in *C*, each kernel covers all the input channels, meaning the operation can be viewed as a convolution with kernel size equal to the input size ($Nks = Nipc$). There are several such kernels generating output channels in parallel (Nop). For dimension *B*, the same kernel is reused by all samples in a mini-batch. This can be viewed as a kernel with one weight moving along the inputs to generate different outputs ($Nopc$).

One benefit of replacing the 4-D integrated operation with the scaled-up 1-D GCONV is the scalability. Fig. 4

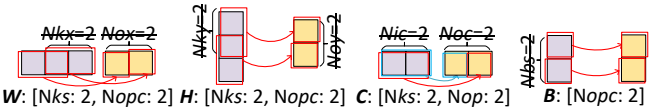

Fig. 5. Describing Convolution Layer in Fig. 2 by GCONV

lists the nested loop for multi-dimension GCONV. Note that the exact same four loops are duplicated (marked as red and blue) for each new dimension, so it is easily inferred that the 4-D GCONV is a nest of 16 loops. Furthermore, we can remove the four loops in dimension *B* to model real-time learning [32], duplicate four loops for time dimension in 3-D CNNs [29] or for vector dimension in Capsule Neural Networks [30], etc. Although the method seems to increase the loops in principle, a certain parameter can be pruned if it takes the default value. Therefore, the GCONV model does not bring any overhead to the effectual number of parameters, as illustrated in Fig. 5. On the contrary, since all the dimensions are perfectly symmetric in terms of the computation and data reuse opportunities, it indeed shrinks our analysis and acceleration design space to just 1-D GCONV with only four loops.

Representability: The proposed model can further shrink the effort-taking study of all kinds of layers in both training and inference into a 1-D GCONV. In a comprehensive analysis, we find that all the layers in modern CNNs can be decomposed into a series of GCONV operations (an example will be given in Section 3.2). And it is future-proof because GCONV can always model a tensor operation by setting the kernel size equal to the input size.

Notice that although the computation pattern of all the layers can be represented by the GCONV parameters, not all of them perform *multiply*-and-*add* operations. Therefore, GCONV operators are introduced aside from the parameters. The four operators of respectively define how the inputs are preprocessed (***pre***); how they are processed by the kernels (***main***); how the partial results within a kernel are reduced (***reduce***); and how the outputs are postprocessed (***post***). The *pre* or *post* operator applies the same processing (e.g., *multiply, and, square* or *LUT*) to each input and output when they are loaded into the convolution engine or sent back to the global buffer. The convolution engine convolves the inputs with the kernels and performs *main* operation (e.g., *square*, *multiply, and* or *add*) between inputs and kernel parameters (which are no longer simply "weights" because the operation is not limited to *multiply*). Some connections in the convolution engine allow *reduction* (e.g., *add* or *compare*) of partial results spatially or temporally. The operators are the same across all the dimensions in a GCONV operation.

Since GCONV does not modify the dataflow, almost all the CNN accelerators can support GCONV computation. The only modification required is that the traditional PEs that only perform *multiply* and *add* should be equipped with more comprehensive *main* and *reduce* functions. This only brings little overhead compared to the expensive dataflow implementation, as will be evaluated in Section 6.4.

3.2 GCONV Chain Generation

With a stack of different functional layers, the end-to-end CNN computation can be converted into a GCONV Chain

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2021.3128159, IEEE Transactions on Computers

TABLE 2
GCONVs for Batch Normalization Layer

GCONV	GCONV Parameters				Input	Kernel Param	Operators				Computation
	B	*C*	*H*	*W*			*pre*	*main*	*reduce*	*post*	
FP											
FP1	[*Nks*: *Nbs*]	[*Nopc*: *Nic*]	[*Nopc*: *Nix*]	[*Nopc*: *Niy*]	L(*l*-1) output				+	× 1/*Nbs*	$\mu = \sum I/Nbs$
FP2	[*Nopc*: *Nbs*]	[*Ng*: *Nic*]	[*Ng*: *Nix*]	[*Ng*: *Niy*]	L(*l*-1) output	FP1 output		−			$t1 = I - \mu$
FP3	[*Nks*: *Nbs*]	[*Nopc*: *Nic*]	[*Nopc*: *Nix*]	[*Nopc*: *Niy*]	FP2 output		^2		+	LUT	$t2 = 1/\sqrt{\sum t1^2/Nbs + \varepsilon}$
FP4	[*Nopc*: *Nbs*]	[*Ng*: *Nic*]	[*Ng*: *Nix*]	[*Ng*: *Niy*]	FP2 output	FP3 output		×			$O = t1 \times t2$
BP											
BP1	[*Nks*: *Nbs*]	[*Ng*: *Nic*]	[*Ng*: *Nix*]	[*Ng*: *Niy*]	L(*l*+1) gradient	FP4 output		×	+	× 1/*Nbs*	$t3 = \sum O \times \boldsymbol{g}O/Nbs$
BP2	[*Nopc*: *Nbs*]	[*Ng*: *Nic*]	[*Ng*: *Nix*]	[*Ng*: *Niy*]	BP1 output	FP4 output		×			$t4 = O \times t3$
BP3	[*Nks*: *Nbs*]	[*Nopc*: *Nic*]	[*Nopc*: *Nix*]	[*Nopc*: *Niy*]	L(*l*+1) gradient				+	× 1/*Nbs*	$t5 = \sum \boldsymbol{g}O/Nbs$
BP4	[*Nopc*: *Nbs*]	[*Ng*: *Nic*]	[*Ng*: *Nix*]	[*Ng*: *Niy*]	L(*l*+1) gradient	BP3 output		−			$t6 = \boldsymbol{g}O - t5$
BP5	[*Ng*: *Nbs*]	[*Nopc*: *Nic*]	[*Nopc*: *Nix*]	[*Nopc*: *Niy*]	BP4 output	BP2 output		−			$t7 = t6 - t4$
BP6	[*Nopc*: *Nbs*]	[*Ng*: *Nic*]	[*Ng*: *Nix*]	[*Ng*: *Niy*]	BP5 output	FP3 output		×			$\boldsymbol{g}I = t7 \times t2$

Nbs: mini-batch size, Nic: number of input channels, Noy/Nox: number of outputs in the H/W dimension per channel, I/O: input/output of the layer, $\boldsymbol{g}I/\boldsymbol{g}O$: gradient of I/O, L(l-1)/(l+1): the last/next layer.

based on producer-consumer relations.

First, here is an example of how to transform the batch normalization (BN) layer in both forward (FP) and back (BP) propagation into a GCONV Chain. In FP, the outputs (O) are the normalization of the inputs (I) over the entire mini-batch:

$$O_i^b = \frac{I_i^b - \mu_i}{\sqrt{\sigma^2{}_i + \varepsilon}} \quad (2),$$

where b denotes the index in B dimension, i iterates in the C, H, W dimensions and ε is a small parameter. μ_i and $\sigma^2{}_i$ are the mean and variance of the mini-batch (the size is *Nbs*):

$$\mu_i = \frac{\sum_{b=0}^{Nbs-1} I_i^b}{Nbs} \quad (3),$$

$$\sigma^2{}_i = \frac{\sum_{b=0}^{Nbs-1} (I_i^b - \mu_i)^2}{Nbs} \quad (4).$$

Table 2 and the GCONVs in bold in Fig. 6 show the GCONV Chain of BN. The FP GCONVs are generated by analyzing the dependencies of Equations (2) to (4). Since both O and σ^2 depend on μ, the calculation of μ is first appended to the chain (**FP1**), which is a reduction in the B dimension. Then $(I - \mu)$ is calculated next (**FP2**) as a GCONV with no reduction but different kernel parameters for each data in C, H, W dimensions. After that, the calculations of σ^2 and O are appended as **FP3** and **FP4** sequentially.

The BP of BN performs the following operation:

$$\boldsymbol{g}I_i^b = \sum_{bb=0}^{bs-1} \frac{\partial O_i^{bb}}{\partial I_i^b} \boldsymbol{g}O_i^{bb}$$

$$= (\underbrace{\underbrace{\boldsymbol{g}O_i^b}_{BP4} - \underbrace{\sum_{bb=0}^{bs-1} \frac{\boldsymbol{g}O_i^{bb}}{bs}}_{BP3}}_{BP5} - \underbrace{O_i^n \underbrace{\sum_{bb=0}^{bs-1} \frac{O_i^{bb} \boldsymbol{g}O_i^{bb}}{bs}}_{BP1}}_{BP2}) / \sqrt{\sigma^2{}_i + \varepsilon} \quad (5),$$

$$\underbrace{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}_{BP6}$$

which is similarly decomposed into six GCONVs. The GCONVs of other layers can also be derived in this way.

Then based on the inter-layer dependencies, we can establish the chain for the entire CNN. Fig. 6 illustrates the conversion of the block in Fig. 1(a) to a GCONV Chain. Like CISC instructions, the original block contains a pile of diverse layers, each requiring complicated customized analysis and optimization. Specially, our proposed technique works as a "*micro code*" layer that translates them into a chain composed of only GCONV operations. It might be noticed that the code density increases in Fig. 6.

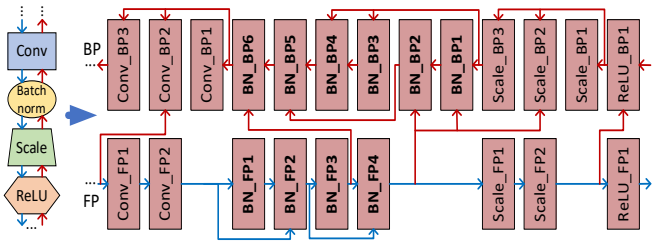

Fig. 6. GCONV Chain of the MobileNet Block in Fig. 1(a)

To this end, we will introduce an operation fusion technique in Section 4.3.

4 GCONV Chain Acceleration

As shown in Fig. 6, the generalization approach proposed in Sections 3 allows diverse operations in CNNs to be converted into a chain of standard GCONV operations. Consequently, the acceleration is no longer layer-specific. Instead, the entire CNN acceleration can be uniformly and systematically achieved by studying the acceleration of a single GCONV operation (Section 4.1) and optimizing the interaction between operations on the chain (Section 4.3).

4.1 Mapping a Single GCONV Operation

To accelerate the GCONV, we need to perform more computation in parallel while increasing the data reuse, which is realized by unrolling and exchanging the order of the nested loop in Fig. 4. The loops can be unrolled both spatially and temporally in an accelerator. Different accelerators have different spatial dimensions to unroll the computation and different memory structures for temporary data storage. Here we build an example on Eyeriss [4], one of the most complicated and self-contained CNN accelerators. The generality of the GCONV mapping will be discussed in Section 4.4.

Accelerator structure: Fig. 7 shows the on-chip structure of Eyeriss. For neatness, we focus on the abstracted spatial and temporal unrolling dimensions while omitting the other implementation details that do not affect GCONV mapping. It contains a $py \times px$ PE array and a global buffer which broadcasts to the PEs. Each PE consists of a *main* and *reduce* (multiply and add in the original work) unit in addition to three local scratchpads for inputs (ILS),

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2021.3128159, IEEE Transactions on Computers

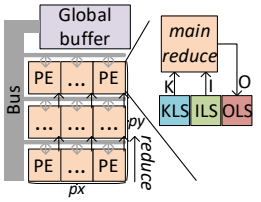

Fig. 7. Eyeriss Structure

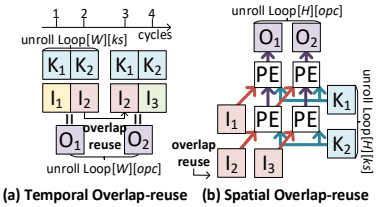

Fig. 8. Eyeriss Overlap-reuse Primitives

Algorithm 1: Algorithm for GCONV Mapping on Eyeriss

Input: GCONV loops of four parameters in four dimensions *loops*; accelerator PE array size $py = 12$, $px = 14$; LS size $ils = 12$, $kls = 224$, $ols = 24$ [4].

Output: two unrolling lists *spatial* and *temporal*.

```
1:  function unrolling (ud, p, d)
2:      uf ← min (remaining resources, loops[d][p])
3:      loops[d][p] ← ceil (loops[d][p]/uf)
4:      remaining resources ← floor (remaining resources/uf)
5:      return uf
6:  function main ()
7:      for d in ["W","H","C","B"] do
8:          if overlap-reuse then
9:              spatial.append (["ks", d, unrolling("py", "ks", d)])
10:             spatial.append (["opc", d, unrolling("px", "opc", d)])
11:         if second overlap-reuse then
12:             temporal.append (["ks", d, unrolling("LS","ks", d)])
13:             temporal.append (["opc", d, loops[d]["opc"]])
14:     for p in ["ks","opc","op","g"] do
15:         for d in ["W","H","C","B"] do
16:             spatial.insert ([p, d, unrolling("py", p, d)])
17:     for p in ["opc","op","ks","g"] do
18:         for d in ["W","H","C","B"] do
19:             spatial.append ([p, d, unrolling("px", p, d)])
20:     for p in ["op","ks", "opc", "g"] do
21:         for d in ["W","H","C","B"] do
22:             temporal.insert ([p, d, unrolling("LS", p, d)])
23:     for p in ["opc","op", "ks", "g"] do
24:         for d in ["W","H","C","B"] do
25:             temporal.append ([p, d, loops[d][p]])
```

ud: accelerator unrolling dimension, *uf*: unrolling factor, LS: local scratchpads.

If *uf* is 1, do not append or insert.

kernel parameters (KLS) and outputs (OLS) to reduce global buffer access.

First, the loops can be spatially unrolled vertically (*py*) or horizontally (*px*) in the PE array. The spatial unrolling determines the parallelization of the computation and the spatial data reuse. The input and kernel parameter parallel-reuses are enabled both horizontally and vertically. The partial results can only be *reduce*d (i.e. output parallel-reuse) vertically thanks to the forwarding links between the rows. Second, the loops can be unrolled temporally so that each PE can reuse the data or *reduce* the partial results locally in LS.

Like many accelerators proposed for convolution layers, the original work of Eyeriss provides overlap-reuse primitives for *W* and *H* dimensions (i.e., row-stationary). As shown in Fig. 8(a), Loop[*W*][*ks*] is unrolled temporally followed by Loop[*W*][*opc*]. This enables the local scratchpads to load only *s* instead of *ks* new inputs each time. In addition, Loop[*H*][*ks*] and Loop[*H*][*opc*] are unrolled in *py* and *px* respectively (Fig. 8(b)). This way, the inputs can be shared diagonally in the PE array. In GCONV, these specially-designed primitives will be allocated to any

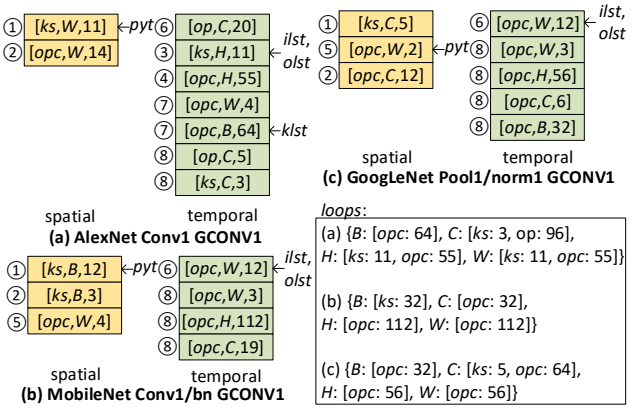

Fig. 9. Example Unrolling Lists (Mini-batch Size is 32)

dimension with overlap-reuse instead of being dedicated to *W* or *H*.

Mapping algorithm: The algorithm for GCONV mapping in Eyeriss is listed in Algorithm 1. The **main** function is a procedure to append unrolling entries to two unrolling lists, i.e., spatial and temporal, until all the loops are unrolled. Each entry in the lists is [*p*, *d*, *uf*], indicating the unrolling factor of parameter *p* in dimension *d* (Loop[*d*][*p*]). The **unrolling** function determines the unrolling factor of an entry by considering the remaining iterations of the loop and the related PE or LS resources (Lines 2 to 4). Here, we explain Algorithm 1 with example mapping results of three different types of layers in Fig. 9, i.e., (a) convolution, (b) batch normalization, (c) local response normalization. Since there are two spatial dimensions, a pointer (*pyt*) is used to point to the tail of the unrolling entries in *py*. Similarly in the temporal unrolling list, three pointers, *ilst*, *olst* and *klst*, point to the last temporal unrolling entries that enable data reuse in ILS, OLS and KLS respectively.

First, to avoid the waste of overlap-reuse primitives, we search for dimensions with overlap-reuse opportunities and unroll *ks* and *opc* in these dimensions in the overlap-reuse primitives (Lines 7 to 13 in Algorithm 1). Note the spatial list is filled before temporal list to maximize parallelism (①②, ③④ in Fig. 9). When performing spatial unrolling, the resources in Lines 2 and 4 are simply the PEs. For temporal unrolling, the entailed LS resources are determined by the amount of data of the unrolled tile, which will be discussed in Section 4.2.

After the overlap-reuse primitives, we further fill the spatial unrolling dimensions (Lines 14 to 19) if there are still spare PEs (⑤). It is important to allow the loops that need a certain function to fill the unrolling dimension with that function first. In Eyeriss, *ks* is first unrolled in *py* to exploit the *reduce* function and *opc* and *op* are first unrolled in *px* to exploit the output bandwidth.

Then the loops are unrolled temporally to fill the local scratchpads to increase data reuses (Lines 20 to 22, unrolling entries ⑥). Here, *op* is first unrolled to reuse the inputs. When a local scratchpad (e.g., *kls*) is full, the loops that reuse this kind of data can still be appended (e.g., ⑦).

When all the resources are exploited, the remaining loops are simply appended (Lines 23 to 25, unrolling entries ⑧). Note that *g* is always unrolled the last because it never manifests any special function or data reuse.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2021.3128159, IEEE Transactions on Computers

TABLE 3
Data Movement for GCONV

Data Type	Reuse	Data Movement
input	$\prod Pop_d$	$\prod(Pg_d \times (Pks_d + Ps_d \times (Popc_d - 1)))$
kernel param	$\prod Popc_d$	$\prod(Pg_d \times Pop_d \times Pks_d)$
output	$\prod Pks_d$	$\prod(Pg_d \times Pop_d \times Popc_d)$

4.2 Modeling the Performance of GCONV Mapping

To enable selection and evaluation of the mapping strategies, this section builds a concise model on how the GCONV mapping results affect the performance and total data movement.

Computation cycles: The total cycles to complete a GCONV can be derived from the spatial unrolling as:

$$Cyc. = \prod_{d\in\{B,C,H,W\}} \prod_{p\in\{ks,opc,op,g\}} ceil(\frac{Np_d}{SP\_Pp_d}) \quad (6),$$

where Pp_d refers to the unrolling factor of parameter p in dimension d and SP means the unrolling in spatial list.

Data movement: The total amount of data for a series of unrollings is related to the data reuse opportunities discussed in Section 3.1. As listed in Table 3, the amounts of inputs, kernel parameters and outputs are independent of *Pop*, *Popc* and *Pks* respectively because of the parallel-reuses. The relation between the input data and *Popc* can be derived using Equation (1), which takes the overlap-reuse into consideration. The total required data is the product of that in all the dimensions.

Therefore, the amount of kernel parameters required by a series of temporal unrollings for each PE can be derived as:

$$TP\_K = \prod_{d\in\{B,C,H,W\}} (TP\_Pg_d \times TP\_Pop_d \times TP\_Pks_d) \quad (7),$$

where *TP* means unrolling in the temporal list. When the required amount of kernel parameters exceeds the capacity of KLS (e.g., the last loop that *klst* points to in Fig. 9), a data movement occurs to load new data to KLS. Therefore, the number of KLS data movements can be derived as:

$$\#KM = \prod_{d\in\{B,C,H,W\}} \prod_{p\in\{ks,opc,op,g\}} out\_klst\_TP\_Pp_d \quad (8),$$

where *out_klst_TP* refers to loops outside the *klst*. Similar to Equation (7), the total kernel parameters required by all the working PEs for each cycle is:

$$SP\_K = \prod_{d\in\{B,C,H,W\}} (SP\_Pg_d \times SP\_Pop_d \times SP\_Pks_d) \quad (9).$$

Based on Equations (7) to (9), the data movement of KLS is:

$$kls\_movement = \#KM \times SP\_K \times in\_klst\_TP\_K \quad (10).$$

The data movement of inputs and outputs and the lower-level memory (e.g., global buffer, off-chip DRAM) can be derived similarly.

4.3 Extending to GCONV Chain Acceleration

Besides the algorithm to map a single GCONV operation to a given accelerator, our system also includes two chain optimizations to overcome the challenges to efficiently accelerate the entire GCONV Chain.

Consistent mapping: The sharing of global buffer requires the consumer to load the intermediate data in the format stored by the producer. For example, in Eyeriss, outputs unrolled in *px* (O_1 and O_2 in Fig. 8(b)) are generated in parallel and can be collected at the same time while the inputs unrolled temporally (I_1 and I_2 in Fig. 8(a)) can be loaded into the local scratchpads in parallel through the data bus. Therefore, the inner *opc*/*op*/*g* loops in *px* unrolling of the producer determine the storage format of intermediate data while the inner *ks*/*opc*/*g* loops of the consumer's temporal unrolling determine the optimal loading format. An inconsistent mapping example is illustrated in Fig. 10. Based on the mapping of the producer (e.g., DenseNet ReLU1 GCONV1) in Fig. 10(a), the buffering format for the intermediate data is shown in Fig. 10(c). However, the mapping of the consumer (e.g., DenseNet Convolution2 GCONV1) in Fig. 10(b) requires loading the inputs in the format in Fig. 10(d), which is not consistent to that in (c).

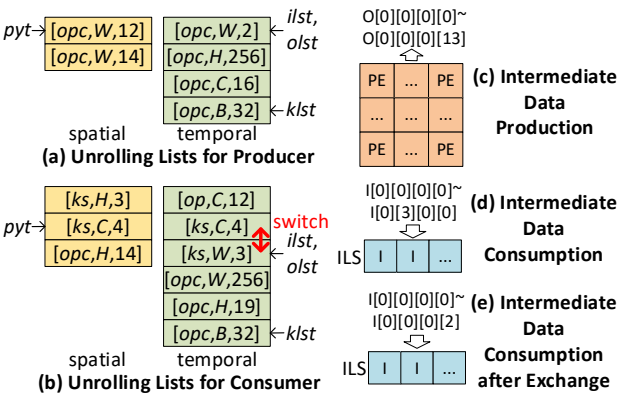

Fig. 10. An Example of Unrolling Loop Exchange

In GCONV Chain, the intermediate data format inconsistency can be simply solved by loop exchange. For instance, in Fig. 10(b), if the unrollings [*ks*, *C*, 4] and [*ks*, *W*, 3] are exchanged, the inputs of the consumer can then be loaded in the format in Fig. 10(e). With the original unrolling, only one input is loaded into ILS per cycle. After the exchange, at least three inputs (determined by the unrolling factor and the width of the buffer) can be loaded in parallel. In practice, we also consider exchange of temporal and spatial unrollings of the same parameter as well as unrollings with different parameters in the same unrolling dimension. Additionally, if there are no appropriate ones in the consumer, we check exchange opportunities in the producer. Note that the unrolling loop exchange does not affect the performance or data movement based on Equations (6) and (10) but significantly reduces the loading time for the consumer. In our experiments, this reduces the data loading latency by up to 3.9x compared to the baseline.

Operation fusion: Operation fusion is commonly adopted to reduce the movement of intermediate data and to fully exploit the memory bandwidth [33]. In GCONV Chain, we also notice an imbalance among the operations in terms of the data/computation ratio. This results in low performance of certain GCONVs with a bottleneck in data loading. Therefore, we apply operation fusion by fusing the GCONVs with no *reduce* operator into the *pre*, *post* or *main* operators of their consumer or producer. For example, GCONV FP2 in Table 2 can be processed as the *post* of FP1 or *pre* of FP3 and FP4. Since the outputs only need to be processed once, fusing to the *post* operator is preferred. After fusion, the *pre* and *post* operators may have more than one parameter and the parameters can be reused in different dimensions. Therefore, to minimize the parameter loading overhead, the consistent mapping also takes the

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2021.3128159, IEEE Transactions on Computers

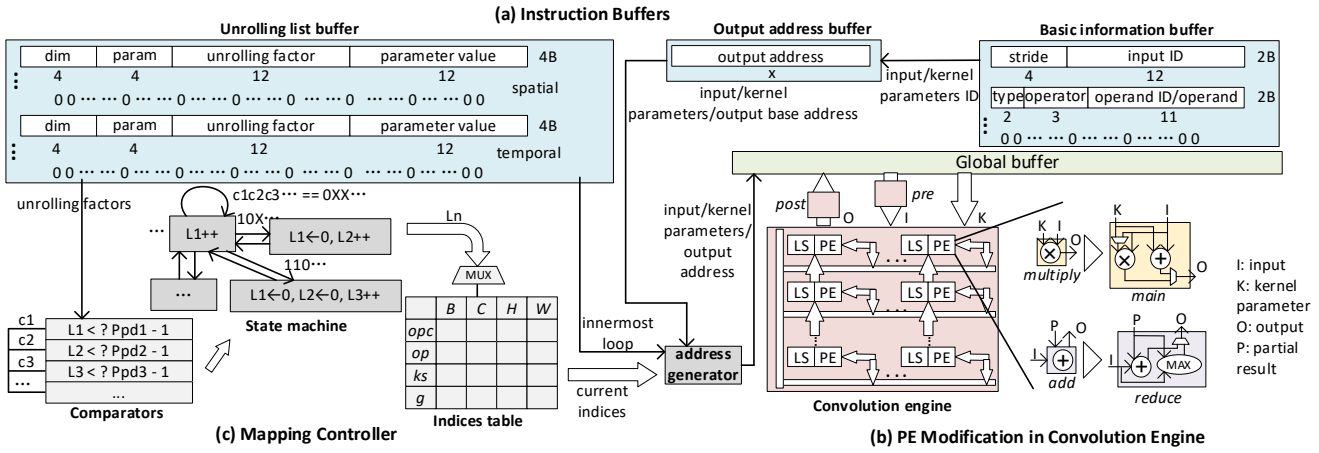

Fig. 11. An Overview of GCONV Chain Implementation

unrolling consistency with the *pre/post* operator into consideration.

The operation fusion reduces the length of GCONV Chain by up to 30%. It also reduces the input movement cost by up to 63%. However, due to the *pre/post* parameter loading, the kernel parameter movement of the global buffer increases. On average, operation fusion improves the performance by 1.1x and decreases the data movement energy by 1.3x.

4.3 Generalizing to Other Accelerators

Although the GCONV Chain acceleration method is exemplified by Eyeriss, it easily generalizes to other accelerators. The performance analysis in Section 4.2 intrinsically applies to all the cases. Therefore, the exact mapping algorithm of a random accelerator only relies on its specific structure for unrolling. In our exploration, all the accelerators manifest both the spatial and temporal unrolling dimensions. The difference lies in the number and functions of the spatial dimensions as well as the capacity and hierarchy of the memory. Among the evaluated accelerators in Table 4, [6][16] possess two spatial dimensions, one with input parallel-reuse and the other with *reduce* but no overlap-reuse; [20] has two spatial dimensions with one for overlap-reuse; and the only spatial dimension in the subsystem of [5] can exploit *reduce* and overlap-reuse at the same time. In terms of memory, most accelerators adopt two-level on-chip storage. For those accelerators with no local scratchpads (e.g. [16]), the sizes of the local scratchpads can be set to 1. In some accelerators, only a certain data type has a local memory (e.g., the input pool in [5]).

Despite the variance in the structure of the accelerators, the underlying mapping philosophy, i.e., to first occupy the spatial dimensions and special functions to maximize the performance and data reuse, always holds. In the evaluation, we follow the mapping strategies in the original works of the baselines, which just slightly changes the priority of the parameters in Lines 7 to 22 of Algorithm 1 in Section 4.1. The two chain optimizations in Section 4.3 do not rely on a certain accelerator either. For a given structure, we just need to recognize the output and input format determining dimensions to guarantee that the inner loops are consistent.

5 GCONV Chain Implementation

To apply GCONV Chain to an existing accelerator, necessary supports are inserted into the computation stack.

First, we implement a compiler that automatically transforms a neural network into a GCONV Chain and then performs optimizations and mapping based on the given accelerator structure. Our compiler is implemented in Python and all the networks and hyperparameters are extracted from Caffe [34] through the Pycaffe interface. For all the CNNs and accelerators, it takes an average of 0.024 seconds to transform and auto-map one layer. This generates a list of GCONV instructions, which are executed by the GCONV-augmented accelerator shown in Fig. 11.

Fig. 11(a) shows the instructions of a GCONV operation. There are three instruction buffers in the system. The basic information buffer stores the stride, operators, input and kernel parameter producer IDs. Considering that some GCONVs do not have *pre*, *main*, *reduce* or *post* operators, the first field of the operator instruction is utilized to indicate the operator type. An all-zero entry delimits the basic information of the GCONVs. For the unrolling list buffer, the first three fields are the unrolling dimension, parameter and unrolling factor respectively, as in Fig. 9, while the last field indicates the argument of the parameter. If the parameter is unrolled more than once, the argument is the sum of all the entries that unroll the same parameter. The unrolling lists in different unrolling dimensions for the GCONVs are also delimited by an all-zero entry. The last instruction buffer stores the address of the output generated by each GCONV, the width of which is determined by the size of the data buffer.

The accelerator is equipped with a set of registers to buffer the stride, parameters, operators and unrolling lists. In the set-up stage of each GCONV, one instruction entry is read from the basic information and unrolling list buffers in each cycle. The decoder translates the instructions dictated by a state machine. During the process, the last entry (e.g. *pyt*, *ilst*) of each dimension and the arguments of the parameters are generated while decoding the unrolling lists. The addresses of input and operands for the operations are derived by indexing the IDs in the output address buffer and the output address is allocated in run-time

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2021.3128159, IEEE Transactions on Computers

TABLE 4
GCONV Chain Implementation Configurations

Category	Accelerator	Configuration	PEs	Local Storage	Global Buffer	Bandwidth
TIP	TPU [16]	64 rows, 64 columns	4096	ILS: 1 per PE, OLS:1 per PE, KLS:1 per PE	I & O: 1.5MB K: 0.25MB	I: 64, O: 64, K: 11
LIP	DNNWeaver (DNNW) [20]	14 PUs, 74 PEs per PU	1036	ILS: 1 per PE, OLS: 1 per PE, KLS: 1 per PU	I & O & K: 8.5kB per PE, K: 8.5kB per PU	I & O & K: 1 for 2 PEs, K: 1 per PU
CIP	Eyeriss (ER) [4]	12 rows, 14 columns	168	ILS: 12 per PE, OLS: 24 per PE, KLS: 224 per PE	I & O: 100kB, K: 8kB	I: 1, O: 4, K: 4
CIP	EagerPruning (EP) [5]	4 subsystems, 512 PEs per subsystem	2048	ILS: 64 per subsystem, OLS: 1 per PE, KLS: 1 per PE	I: 1.5MB, O: 1.5MB, K: 1.5MB	I: 32 per subsystem, O: 32 per subsystem, K: 32 per subsystem
CIP	NLR [6]	Tm = 64, Tn = 7	448	ILS: 1 per Tn, OLS: 1 per Tm, KLS: 1 per PE	I & K: 1.5MB, O: 0.75MB	I & K: 7, O: 64

based on the current data buffer occupation and the size of the output. To eliminate the possible delay, instruction loading and decoding are overlapped with the processing of the previous GCONV.

As mentioned in Section 3.1, GCONV does not change the inherent connections in the convolution engine. The only modification is to replace the original *multiply* and *add* functions with comprehensive *main* and *reduce* functions, as shown in Fig. 11(b). As in [16], we deploy 8-bit data, 16 bits for the results of *main* operator and 32 bits for the results of *reduce* and *post* operators. The precision-sensitive steps are fused into *post* operators. During GCONV processing, the loop iterations are carried out by a state machine, as shown in Fig. 11(c). Since the unrolling lists are not fixed, it is impossible to use a predefined state machine. Instead, the transition conditions are set as the results of comparison between the unrolling factors and the counters. A 16:1 MUX is adopted to increase the index of the corresponding parameter. The address generator generates the offset of the data based on the index and data storage layout. The layout generally follows the order of the indices in Fig. 4. The indices on the right lie in inner loops of the data layout. However, as discussed in Section 4.3, the producer can generate and store data in any dimension in parallel. Therefore, based on the unrolling list of the producer, the layout is adjusted to move a certain dimension to the front. Note that this does not change the address calculation logic but just the parameters. Then based on the address, the data loading module loads the data and feeds them to the PEs through the data bus.

6 Evaluation

6.1 Baselines and Benchmarks

For a comprehensive evaluation, we include all three types of CNN accelerators discussed in Section 2.3, as listed in Table 4. [4]–[6] baseline cases adopt the PE and memory configurations as in the original works. Note that we perform dense computation on EP to focus on the hardware acceleration. [16] is proposed as a datacenter-level design, so we scale down its basic block by 4×4 to match the other accelerators. [20] generates a customized accelerator for each CNN. We adopt the configuration of AlexNet, the only benchmark we share, on the moderate FPGA Altera Stratix V SGSD5. For GCONV Chain implementation, all the layers are converted into GCONV operations and automapped to the convolution engine (or matrix functional unit in TIPs) of the accelerators. Since DNNW allocates computation resources to some other dedicated functional units, which will be idle in GCONV Chain processing, the convolution engine of the GCONV Chain implementation is scaled up so that they have the same number of PEs and total bandwidth as the baselines. Columns 3-7 of Table 4 summarize the GCONV Chain configurations.

For the benchmarks, we evaluate the seven CNNs in Table 1. Note that ZFFR, CapNN and C3D are not evaluated on baseline DNNW and C3D is not evaluated on all the CIP baselines since on-chip acceleration of the functional layers in these networks is unclear in the original papers. In the experiment, we focus on the training of CNNs, which includes the computation in inference and provides more insights.

6.2 Methodologies

To demonstrate the three benefits brought by GCONV Chain discussed in Section 1, we study the speedup, overhead, energy efficiency and whole-life cost of GCONV Chain. Specifically, Section 6.3 evaluates the speedup of GCONV Chain over baselines to show that it can be applied to any accelerator. Sections 6.5 and 6.6 compare the energy efficiency of GCONV Chain-armed CIPs with TIPs and LIPs to show its potential in low-cost CNN acceleration.

We develop a simulator to evaluate the performance and data movement based on the model proposed in Section 4.2, which is validated on a cycle-accurate basis. To get the area and energy estimation, we prototype the accelerators and synthesize the RTL using Synopsys Design Compiler and simulate the memory with CACTI [35]. All the accelerators run at 700MHz. In baseline CIPs, only the traditional layers mentioned in Section 2.2 are processed on-chip while the others are offloaded to an ARM A53 CPU through PCIe 4.0. Computations allocated to different functional units or the host are processed in a pipeline.

6.3 Speedup

The end-to-end speedup brought by GCONV Chain comes from two aspects: (1) it eliminates the inefficiencies of the baselines in terms of processing the non-traditional layers; (2) for the most computation-intensive convolution layers, GCONV Chain can still improve the performance thanks to its flexible mapping.

To implicate the inefficiencies of the baselines, Fig. 12 first shows the latency breakdown of them. Among the baseline accelerators, TPU and DNNW suffer from pipeline bubbles with considerable time only running either the

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2021.3128159, IEEE Transactions on Computers

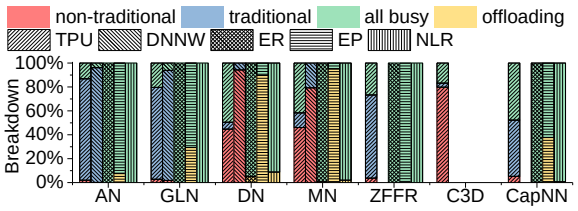

Fig. 12. Baseline Latency Breakdown

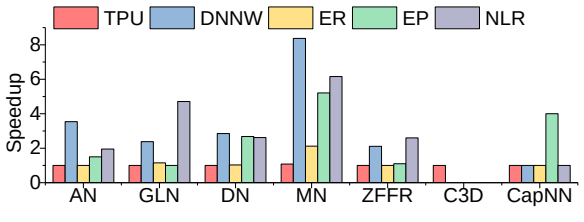

Fig. 13. Convolution Layers Speedup

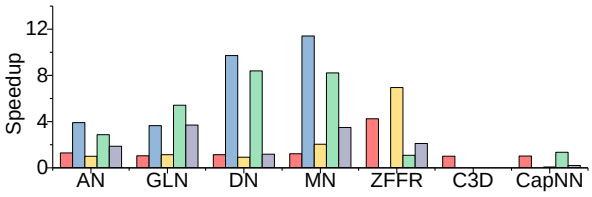

Fig. 14. End-to-end Speedup

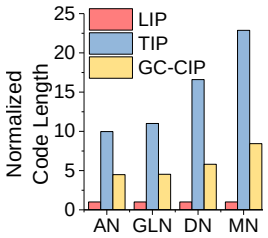

Fig. 15. Code Length Comparison

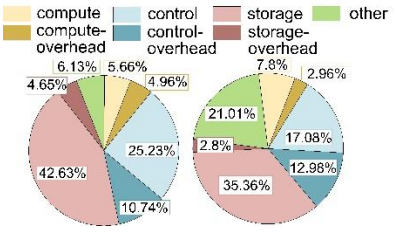

Fig. 16. Area Breakdown

Fig. 17. Power Breakdown

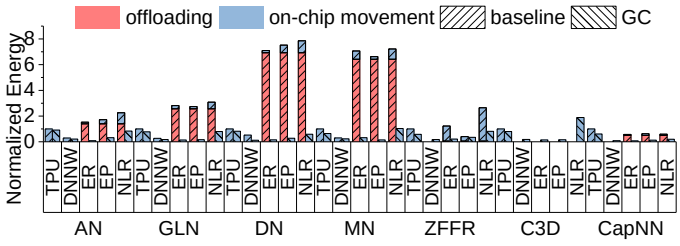

Fig. 18. On-chip and Offloading Data Movement Energy

traditional or non-traditional layers (computation-traditional or computation-non-traditional in Fig. 12). The runtime that all the components are busy (all-busy) only accounts for 31% and 2% in TPU and DNNW respectively. The utilization is higher in TPU because it accelerates fine-grained tensor operations while the instructions in DNNW are more complex. EP suffers the most from the offloading (43% of runtime on average) because it has the highest on-chip performance. While ER and NLR can overlap the offloading by computation to some extent, the offloading power is not negligible as will be shown in Fig. 18. In terms of each CNN, the offloading latency is more severe in recent CNNs with more non-traditional layers (e.g., DN, MN). However, CNNs with non-traditional layers highly concatenated (e.g., ZFFR, C3D, CapNN) suffer less from offloading.

Fig. 13 shows the speedup of the convolution layers to demonstrate the effectiveness of GCONV mapping. In all the cases, the performance of GCONV Chain is no worse than the baselines. In MN, where the feature maps unrolling in the baselines is useless for depthwise convolution, the speedup is salient. GCONV Chain also significantly speeds up the convolution layers in baseline NLR, which only unrolls the input and output feature maps. The speedup over baseline TPU and ER are low because they explore flexible unrolling strategies. EP is similarly flexible as ER but the huge PE array makes the baseline mapping less effective. Fortunately, GCONV Chain manages to improve its performance.

When it comes to the end-to-end CNN acceleration including all the traditional and non-traditional layers, Fig. 14 shows the speedup of GCONV Chain to the baselines. The results show that GCONV Chain speeds up the baselines by up to 8.2x and an average of 3.4x among all the accelerators. The speedup of DN and MN on DNNW and EP are high because their baselines suffer the most from the pipeline bubbles and offloading. The speedup of CapNN on ER and NLR is low because their on-chip computing power cannot compare to that of A53.

6.4 Overhead

We aim to compare the total cost of GCONV Chain-armed CIPs (GC-CIPs) with LIPs and TIPs, so this section focuses on the overheads brought by GCONV Chain to CIPs. Fig. 15 compares the average code length of GC-CIPs with LIPs and TIPs. On average, GC-CIPs instructions are 5.8x longer than LIPs because LIPs have only one instruction for each layer. TIPs only process basic matrix or vector algorithms, so control operations are needed when the computation cannot be mapped to only one matrix/vector operation. In addition, they require load instructions while LIPs and GC-CIPs load data implicitly. Therefore, their code density is the worst (2.6x worse than GC-CIPs).

Fig. 16 and 17 list the overhead of GCONV Chain in the area and the average power breakdown of Eyeriss. The storage overhead refers to the storage for the instruction buffers in Fig. 11(a) and the compute overhead corresponds to the PE modification in Fig. 11(b). The control overhead includes all the required signals in Fig. 11(a)(b) and the controller in Fig. 11(c). In total, GCONV Chain brings 20% area and 19% power consumption overhead. This is acceptable considering the speedup and reduction in data movement.

6.5 Energy Efficiency

In CNN accelerators, it is widely recognized that the data movement dominates the energy efficiency [19]. The measurement in Fig. 18 includes the on-chip global buffer movements and offloading and reloading related power normalized to the baseline of TPU. The off-chip data movement is not considered because GCONV Chain does not substantially affect the off-chip data access in our experiments. As shown, although the on-chip data movement reduction brought by GCONV Chain is not significant, it eliminates

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2021.3128159, IEEE Transactions on Computers

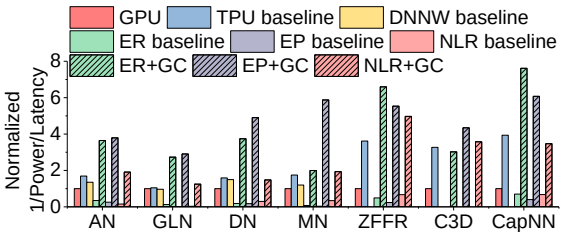

Fig. 19. Energy Efficiency (Iso-power Performance)

the costly offloading and reloading of non-traditional layers in CIPs. Compared with TIPs, GC-CIPs that explore more data reuses have the lowest data movement (16% and 22% in ER and EP). Note that NLR does not have low on-chip data movement because it does not exploit any overlap-reuse.

Fig. 19 further shows the normalized overall energy efficiencies of the GC-CIPs, TIP, LIP and a state-of-the-art GPU, i.e., NVIDIA Tesla V100. Equipped with GCONV Chain, the CIP accelerators with overlap-reuse (i.e., ER and EP) overcome the inefficiency in the baselines (37.6x on average) and show a promising edge over TIP (up to 3.4x, 2.1x on average), LIP (up to 4.9x, 3.0x on average) and GPU (up to 7.6x, 4.5x on average).

6.6 Whole-life Cost

Last but not the least, we compare the whole-life costs of TIP, LIP and GC-CIP. Fig. 20 shows the development costs as a sum of hardware/software non-recurring expenses (NRE) and update costs. Based on the complexity level of the accelerator implementation, the hardware NRE of TIPs, GC-CIPs and LIPs are quoted as 152K, 165K and 220K USDs [36]. Then in each update, LIPs require 200K USDs on the new hardware design. The software NRE and update costs are calculated using the latest salary [37][38] and lines of code in our prototype compiler. Although GC-CIPs consume more in the hardware than TIPs, the software development is cheaper due to code generation complexity. This gap widens with more updates and 60K additional USDs are consumed for development of TIPs than GC-CIPs after ten updates.

Users who invest in the accelerators need to pay the capital expenses (CAPEX) for the device purchase and annual update and the utility as operating expenses (OPEX). Fig. 21 shows the total costs of ownership for the ASIC version of the three types of accelerators as well as FPGA LIPs and GPUs, which are popular choices for CNN acceleration. The CAPEX of FPGA and the ASICs are scaled to meet the performance of GPU and the operating utility is calculated assuming the devices are always working at the average utility rate in US [39]. As observed, the GPU, FPGA and ASIC LIPs with high CAPEX [36][40][41] are not the best choices for pure CNN acceleration. Thanks to the high energy efficiency of convolution customized dataflows, GC-CIPs win the most whole-life efficient CNN accelerators by costing 45% less than TIPs after just three years and 65% less after ten years.

7 Related Work

Besides the accelerators discussed in Section 2.3, there are

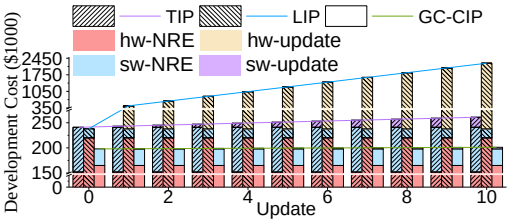

Fig. 20. Development Cost

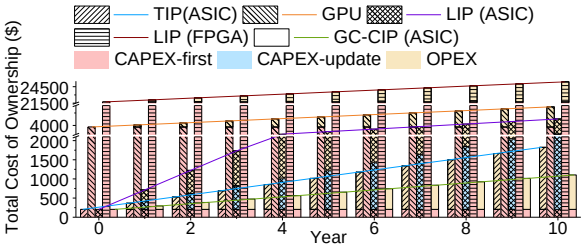

Fig. 21. Total Cost of Ownership

several works trying to efficiently process the non-traditional computation in CNNs. [7] infuses the adder tree with pooling functions and [9] adds local response normalization layer support but the other layers are still left behind. [42] introduces a method to break down the batch normalization into two parts and fuse the computation to the next and last layers in the CNNs, which is orthogonal to our work and can be adopted when optimizing the GCONV Chain. [43] introduces batch size as a tunable parameter in CNN accelerators to make up for the lack of parallelism in traditional convolution acceleration but it does not systematically accelerate all the non-traditional layers. [33] is proposed to assist mapping neural networks defined in any framework to any hardware. However, it currently only supports matrix/vector operations and commercial general-purpose processors. [44][45] propose models to explore the mapping design space of neural network accelerators but they do not provide a systematic solution to the end-to-end CNN acceleration.

8 Conclusion

This paper has addressed a highly critical but generally overlooked challenge: *the efficient and cost-effective acceleration of the diverse end-to-end CNN computation*. To exploit the reuse opportunities in CNNs and to avoid resource underutilization and costly upgrade brought by allocating dedicated hardware to the non-traditional layers, we proposed a general convolution model and generalizing the diverse computations in CNNs into GCONV Chain. By generalization, the end-to-end GCONV Chain can be efficiently processed by existing accelerators customized for convolution with low-overhead hardware support. Our evaluation shows that GCONV Chain manifests great potential in accelerating the CNNs with *high performance, energy efficiency and low whole-life cost*.

Acknowledgment

We appreciate the Department of Electrical and Computer Engineering, University of Florida for the support.

REFERENCES

[1] C. E. Floyd, J. Y. Lo, A. J. Yun, D. C. Sullivan, and P. J. Kornguth, "Prediction of breast cancer malignancy using an artificial neural network," Cancer, 1994.

[2] H. A. Rowley, S. Baluja, and T. Kanade, "Neural Network-Based Face Detection," TPAMI, 1998.

[3] M. Bojarski et al., "End to End Learning for Self-Driving Cars," arXiv:1604.07316, 2016.

[4] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks," IEEE J. Solid-State Circuits, vol. 1, 2016.

[5] J. Zhang, X. Chen, M. Song, and T. Li, "Eager Pruning: Algorithm and Architecture Support for Fast Training of Deep Neural Networks," in Proceedings of the 46th International Symposium on Computer Architecture (ISCA), 2019, pp. 292–303.

[6] 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 (FPGA), 2015, pp. 161–170.

[7] H. Kwon, A. Samajdar, and T. Krishna, "MAERI: Enabling Flexible Dataflow Mapping over DNN Accelerators via Reconfigurable Interconnects," in Proceedings of the Twenty-Third International Conference on Architectural Support for Programming (ASPLOS), 2018, vol. 53, no. 2, pp. 461–475.

[8] W. Lu, G. Yan, J. Li, S. Gong, Y. Han, and X. Li, "FlexFlow: A Flexible Dataflow Accelerator Architecture for Convolutional Neural Networks," in Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA), 2017, pp. 553–564.

[9] Y. Chen et al., "DaDianNao: A Machine-Learning Supercomputer," in Proceedings of the Annual International Symposium on Microarchitecture (MICRO), 2015, vol. 2015-Janua, no. January, pp. 609–622.

[10] Z. Du et al., "ShiDianNao: Shifting Vision Processing Closer to the Sensor," in Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA), 2015.

[11] K. Hegde, J. Yu, R. Agrawal, M. Yan, M. Pellauer, and C. W. Fletcher, "UCNN: Exploiting Computational Reuse in Deep Neural Networks via Weight Repetition," in Proceedings of the International Symposium on Computer Architecture (ISCA), 2018, pp. 674–687.

[12] H. Lu, X. Wei, N. Lin, G. Yan, and X. Li, "Tetris: Re-Architecting Convolutional Neural Network Computation for Machine Learning Accelerators," in Proceedings of the International Conference on Computer-Aided Design, Digest of Technical (ICCAD), 2018, pp. 1–8.

[13] A. G. Howard et al., "MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications," arXiv:1704.04861, 2017.

[14] S. Ioffe and C. Szegedy, "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift," in Proceedings of the 32nd International Conference on Machine Learning (ICML), 2015, vol. 1, pp. 448–456.

[15] J. Frankle, D. J. Schwab, and A. S. Morcos, "Training BatchNorm and Only BatchNorm: On the Expressive Power of Random Features in CNNs," arXiv:2003.00152, Feb. 2020.

[16] N. P. Jouppi et al., "In-Datacenter Performance Analysis of a Tensor Processing Unit," in Proceedings of the International Symposium on Computer Architecture (ISCA), 2017, vol. Part F1286, pp. 1–12.

[17] S. Chetlur et al., "cuDNN: Efficient Primitives for Deep Learning," arXiv:1410.0759, 2014.

[18] S. Liu et al., "Cambricon: An Instruction Set Architecture for Neural Networks," in Proceedings of the 43rd International Symposium on Computer Architecture (ISCA), 2016, pp. 393–405.

[19] S. Han et al., "EIE: Efficient Inference Engine on Compressed Deep Neural Network," ACM SIGARCH Comput. Archit. News, vol. 44, no. 3, pp. 243–254, Oct. 2016.

[20] H. Sharma et al., "From High-Level Deep Neural Models to FPGAS," in Proceedings of the Annual International Symposium on Microarchitecture (MICRO), 2016, vol. 2016-Decem.

[21] Y. S. Shao et al., "Simba: Scaling Deep-learning Inference with Multi-Chip-Module-Based Architecture," in Proceedings of the Annual International Symposium on Microarchitecture (MICRO), 2019, pp. 14–27.

[22] S. Venkataramani et al., "Scaledeep: A Scalable Compute Architecture for Learning and Evaluating Deep Networks," in Proceedings of the International Symposium on Computer Architecture (ISCA), 2017, vol. Part F1286, pp. 13–26.

[23] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," Commun. ACM, vol. 60, no. 6, pp. 84–90, Jun. 2017.

[24] Y. LeCun et al., "Handwritten digit recognition: applications of neural network chips and automatic learning," Commun. Mag., 1989.

[25] C. Szegedy et al., "Going Deeper with Convolutions," in Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition, 2015, vol. 07-12-June, pp. 1–9.

[26] G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger, "Densely Connected Convolutional Networks," in Proceedings of the 30th Conference on Computer Vision and Pattern Recognition (CVPR), 2017, vol. 2017-Janua, pp. 2261–2269.

[27] S. Ren, K. He, R. Girshick, and J. Sun, "Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks," IEEE Trans. Pattern Anal. Mach. Intell., vol. 39, no. 6, pp. 1137–1149, Jun. 2017.

[28] M. D. Zeiler and R. Fergus, "Visualizing and Understanding Convolutional Networks," in Proceedings of the European Conference on Computer Vision (ECCV), 2014, vol. 8689 LNCS, no. PART 1, pp. 818–833.

[29] D. Tran, L. Bourdev, R. Fergus, L. Torresani, and M. Paluri, "Learning Spatiotemporal Features with 3D Convolutional Networks," in Proceedings of the International Conference on Computer Vision (ICCV), 2015, pp. 4489–4497.

[30] S. Sabour, N. Frosst, and G. E. Hinton, "Dynamic Routing Between Capsules," Adv. Neural Inf. Process. Syst., vol. 2017-Decem, pp. 3857–3867, Oct. 2017.

[31] M. Song, J. Zhang, H. Chen, and T. Li, "Towards Efficient Microarchitectural Design for Accelerating Unsupervised GAN-Based Deep Learning," in Proceedings of the International Symposium on High Performance Computer Architecture (HPCA), 2018, pp. 66–77.

[32] S. C. H. Hoi, D. Sahoo, J. Lu, and P. Zhao, "Online Learning: A Comprehensive Survey," arXiv:1802.02871, 2018.

[33] T. Chen et al., "TVM: An Automated End-to-End Optimizing Compiler for Deep Learning," in Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation, 2018, pp. 578–594.

[34] Y. Jia et al., "Caffe: Convolutional Architecture for Fast Feature Embedding," in Proceedings of the 2014 ACM Conference on Multimedia, 2014, pp. 675–678.

[35] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "CACTI 6.0: A Tool to Model Large Caches."

[36] "Custom ASIC Cost Calculator - Sigenics." [Online]. Available: https://www.sigenics.com/page/custom-asic-cost-calculator?gclid=Cj0KCQjw2or8BRCNARIsAC_ppy-bUluAr8VWwRqO73cqrcugkB3zVH_EN54A6SE7t-bkBvmii32QGXF0aAlJKEALw_wcB.

[37] "Apple Software Engineering Salaries | Glassdoor." [Online]. Available: https://www.glassdoor.com/Hourly-Pay/Apple-Software-Engineering-Hourly-Pay-E1138_D_KO6,26.htm.

[38] "Better Embedded System SW." [Online]. Available:

https://betterembsw.blogspot.com/2010/05/only-10-lines-of-code-per-day-really.html.

[39] "Electricity Rates by State (Updated September 2020) – Electric Choice." [Online]. Available: https://www.electricchoice.com/electricity-prices-by-state/.

[40] "Xilinx Evaluation Boards." [Online]. Available: https://www.xilinx.com/products/boards-and-kits/see-all-evaluation-boards.html.

[41] "Welcome to the Official NVIDIA Store." [Online]. Available: https://www.nvidia.com/en-us/shop/.

[42] W. Jung, D. Jung, B. Kim, S. Lee, W. Rhee, and J. H. Ahn, "Restructuring Batch Normalization to Accelerate CNN Training," arXiv:1807.01702, 2019.

[43] Y. Shen, M. Ferdman, and P. Milder, "Escher: A CNN Accelerator with Flexible Buffering to Minimize Off-Chip Transfer," in Proceedings of the International Symposium on Field-Programmable Custom Computing Machines (FCCM), 2017, pp. 93–100.

[44] A. Parashar et al., "Timeloop: A Systematic Approach to DNN Accelerator Evaluation," in Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS), 2019.

[45] X. Yang et al., "Interstellar: Using Halide's Scheduling Language to Analyze DNN Accelerators," in Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2020, pp. 369–383.

Jiaqi Zhang received the B.S. degree in Communication Engineering from Beijing Jiaotong University in 2016. She is currently a Ph.D. candidate in the Department of Electrical and Computer Engineering, University of Florida. Her research interests lie in software and hardware acceleration of emerging algorithms and applications including machine learning and IoT.

Xiangru Chen received the B.S. degree in Electronic Information Engineering from Shandong University in 2016 and M.S. degree in Electrical and Computer Engineering from University of Florida in 2018. He is currently pursuing a Ph.D. degree in the Department of Electrical and Computer Engineering, University of Florida. His research focuses on the architecture support for ML applications.

Sandip Ray is an Endowed IoT Term Professor at the Department of Electrical and Computer Engineering, University of Florida. His research involves developing correct, dependable, secure, and trustworthy computing through cooperation of specification, synthesis, architecture and validation technologies. He focuses on next generation computing applications, including IoT, autonomous automotive systems, etc. Before joining University of Florida, he was a Senior Principal Engineer at NXP Semiconductors, where he led the R&D on security architecture and validation of hardware platforms for automotive and IoT applications. Prior to that, he was a Research Scientist at Intel Strategic CAD Labs, where he led research on validation technologies for security and functional correctness of SoC designs. Dr. Ray is the author of three books and over 90 publications in international journals and conferences. He has a Ph.D. from University of Texas at Austin and is a Senior Member of IEEE.