Optimizing the Convolution Operation to Accelerate Deep Neural Networks on FPGA

Yufei Ma, *Student Member, IEEE*, Yu Cao, *Fellow, IEEE*, Sarma Vrudhula, *Fellow, IEEE*,
and Jae-sun Seo, *Senior Member, IEEE*

***Abstract*— As convolution contributes most operations in convolutional neural network (CNN), the convolution acceleration scheme significantly affects the efficiency and performance of a hardware CNN accelerator. Convolution involves multiply and accumulate operations with four levels of loops, which results in a large design space. Prior works either employ limited loop optimization techniques, e.g., loop unrolling, tiling, and interchange, or only tune some of the design variables after the accelerator architecture and dataflow are already fixed. Without fully studying the convolution loop optimization before the hardware design phase, the resulting accelerator can hardly exploit the data reuse and manage data movement efficiently. This paper overcomes these barriers by quantitatively analyzing and optimizing the design objectives (e.g., memory access) of the CNN accelerator based on multiple design variables. Then, we propose a specific dataflow of hardware CNN acceleration to minimize the data communication while maximizing the resource utilization to achieve high performance. The proposed CNN acceleration scheme and architecture are demonstrated by implementing end-to-end CNNs including NiN, VGG-16, and ResNet-50/ResNet-152 for inference. For VGG-16 CNN, the overall throughputs achieve 348 GOPS and 715 GOPS on Intel Stratix V and Arria 10 FPGAs, respectively.**

***Index Terms*— Accelerator architectures, convolutional neural networks (CNNs), field-programmable gate array (FPGA), neural network hardware.**

I. Introduction

THE field-programmable gate arrays (FPGA) are fast becoming the platform of choice for accelerating the inference phase of deep convolutional neural networks (CNNs). In addition to their conventional advantages of reconfigurability and shorter design time over application-specific integrated circuits (ASICs) [20], [21] to catch up with the rapid evolving of CNNs, FPGA can realize low latency inference with competitive energy efficiency (∼10–50 GOP/s/W) when compared to software implementations on multicore processors with GPUs [10], [12], [13], [17]. This is due to the fact that modern FPGAs allow customization of the architecture and can exploit the availability of hundreds to thousands of on-chip DSP blocks. However, significant challenges remain in mapping CNNs onto FPGAs. The state-of-the-art CNNs require a large number ($>$1 billion) of computationally intensive task (e.g., matrix multiplications on large numbers), involving a very large number of weights ($>$50 million) [4], [5]. Deep CNN algorithms have tens to hundreds of layers, with significant differences between layers in terms of sizes and configurations. The limited computational resources and storage capacity on FPGA make the task of optimal mapping of CNNs (e.g., minimizing latency subject to energy constraints or vice versa) a complex and multidimensional optimization problem. The high cost of off-chip communication is another major impediment to achieving higher performance and lower energy. In fact, the energy cost associated with the large amount of data movements and memory accesses often exceeds the energy consumption of the computations [8], [20]. For these reasons, energy-efficient hardware acceleration of CNNs on a FPGA requires simultaneous maximization of resource utilization and data reuse, and minimization of data communication.

More than 90% of the operations in a CNN involve convolutions [2]–[4]. Therefore, it stands to reason that acceleration schemes should focus on the management of parallel computations and the organization of data storage and access across multiple levels of memories, e.g., off-chip dynamic random access memory (DRAM), on-chip memory, and local registers. In CNNs, convolutions are performed by four levels of loops that slide along both kernel and feature maps as shown in Fig. 1. This gives rise to a large design space consisting of various choices for implementing parallelism, sequencing of computations, and partitioning the large data set into smaller chunks to fit into on-chip memory. These problems can be handled by the existing loop optimization techniques [6], [9], such as loop unrolling, tiling, and interchange. Although some CNN accelerators have adopted these techniques [9], [11], [13], [19], the impact of these techniques on design efficiency and performance has not been systematically and sufficiently studied. Without fully studying the loop operations of convolutions, it is difficult to efficiently customize the dataflow and architecture for high-throughput CNN implementations. This paper aims to address

Manuscript received October 27, 2017; revised February 3, 2018; accepted March 6, 2018. Date of publication April 3, 2018; date of current version June 26, 2018. This work was supported in part by the NSF I/UCRC Center for Embedded Systems through NSF under Grant 1230401, Grant 1237856, Grant 1701241, Grant 1361926, Grant 1535669, Grant 1652866, and Grant 1715443; and in part by the Intel Labs, and in part by the Samsung Advanced Institute of Technology. *(Corresponding author: Yufei Ma.)*

Y. Ma, Y. Cao, and J.-s. Seo are with the School of Electrical, Computer, and Energy Engineering, Arizona State University, Tempe, AZ 85287 USA (e-mail: yufeima@asu.edu; yu.cao@asu.edu; jaesun.seo@asu.edu).

S. Vrudhula is with the School of Computing, Informatics, Decision Systems Engineering, Arizona State University, Tempe, AZ 85287 USA (e-mail: vrudhula@asu.edu).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TVLSI.2018.2815603

1063-8210 © 2018 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

Authorized licensed use limited to: University of Louisiana at Lafayette. Downloaded on October 19,2020 at 23:35:00 UTC from IEEE Xplore. Restrictions apply.

```
for ($no$ = 0; $no$ < Nof; $no$ ++)  ⟶ Loop-4
    for ($y$ = 0; $y$ < Noy; $y$ ++)     } ⟶ Loop-3
        for ($x$ = 0; $x$ < Nox; $x$ ++) }
            for ($ni$ = 0; $ni$ < Nif; $ni$ ++)  ⟶ Loop-2
                for ($ky$ = 0; $ky$ < Nky; $ky$ ++)     } ⟶ Loop-1
                    for ($kx$ = 0; $kx$ < Nkx; $kx$ ++) }
                        $pixel_L(no; x, y) \mathrel{+}= pixel_{L-1}(ni; S \times x + kx, S \times y + ky) \times weight_{L-1}(ni, no; kx, ky)$;
            $pixel_L(no; x, y) = pixel_L(no; x, y) + bias(no)$;
```

Fig. 1. Four levels of convolution loops, where L denotes the index of convolution layer and S denotes the sliding stride [15].

these shortcomings. Specifically, the main contributions of this paper include the following.

1) We provide an in-depth analysis of the three loop optimization techniques for convolution operations and use corresponding design variables to numerically characterize the acceleration scheme.
2) The design objectives of CNN accelerators (e.g., latency, memory) are quantitatively estimated based on the configurations of the design variables.
3) An efficient convolution acceleration strategy and dataflow is proposed aimed at minimizing data communication and memory access.
4) A data router is designed to handle different settings for convolution sliding operations, e.g., strides and zero paddings, especially for highly irregular CNNs.
5) A corresponding hardware architecture is designed that fully utilizes the computing resources for high performance and efficiency, which is uniform and reusable for all the layers.
6) The proposed acceleration scheme and architecture is validated by implementing large-scale deep CNN algorithms, NiN [3], VGG-16 [4], and ResNet-50/ResNet-152 [5] for image recognition [1], on two Intel FPGAs. The proposed accelerators achieve end-to-end inference throughput of 715 GOPS on Arria 10 and 348 GOPS on Stratix V, respectively, using a batch size of 1.

The rest of this paper is organized as follows. Section II identifies the key design variables that are used to numerically characterize the loop optimization techniques. Section III contains a quantitative analysis of hardware accelerator objectives. Section IV describes the acceleration schemes used in some of the recent state-of-the-art CNN accelerators. Section V presents the optimized acceleration scheme with specific design variables. A corresponding dataflow and architecture is proposed in Section VI. Section VII analyzes the experimental results and compares with prior works. Conclusions are presented in Section VIII.

II. Acceleration of Convolution Loops

A. General CNN Acceleration System

Recently reported CNN algorithms involve a large amount of data and weights. For them, the on-chip memory is insufficient to store all the data, requiring gigabytes of external memory. Therefore, a typical CNN accelerator consists of three levels of storage hierarchy: 1) external memory; 2) on-chip buffers; and 3) registers associated with the processing engines (PEs), as shown in Fig. 2. The basic flow is to fetch data from external memory to on-chip buffer, and then feed them into registers and PEs. After the PE computation completes, results are transferred back to on-chip buffers and to the external memory if necessary, which will be used as input to the subsequent layer.

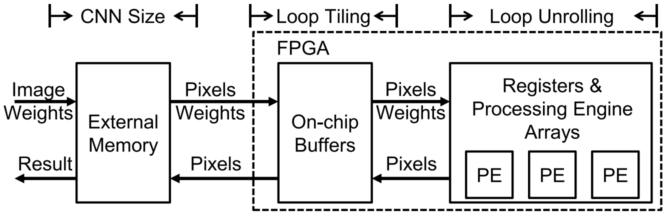


Fig. 2. Three levels of general hardware CNN accelerator hierarchy.

B. Convolution Loops

Convolution is the main operation in CNN algorithms, which involves 3-D multiply-and-accumulate (MAC) operations of input feature maps and kernel weights. Convolution is implemented by four levels of loops as shown in the pseudocodes in Fig. 1 and illustrated in Fig. 3. To efficiently map and perform the convolution loops, three loop optimization techniques [6], [9], namely, loop unrolling, loop tiling, and loop interchange, are employed to customize the computation and communication patterns of the accelerator with three levels of memory hierarchy.

C. Loop Optimization and Design Variables

As shown in Fig. 3, multiple dimensions are used to describe the sizes of the feature and kernel maps of each convolution layer for a given CNN. The hardware design variables of loop unrolling and loop tiling will determine the acceleration factor and hardware footprint. All dimensions and variables used in this paper are listed in Table I.

The width and height of one kernel (or filter) window is described by (*Nkx*, *Nky*). (*Nix*, *Niy*) and (*Nox*, *Noy*) define the width and height of one input and output feature map (or channel), respectively. *Nif* and *Nof* denote the number of input and output feature maps, respectively. The loop unrolling design variables are (*Pkx*, *Pky*), *Pif*, (*Pox*, *Poy*), and *Pof*, which denote the number of parallel computations. The loop tiling design variables are (*Tkx*, *Tky*), *Tif*, (*Tox*, *Toy*), and *Tof*, which represent the portion of data of the four loops stored in on-chip

Authorized licensed use limited to: University of Louisiana at Lafayette. Downloaded on October 19,2020 at 23:35:00 UTC from IEEE Xplore. Restrictions apply.

TABLE I
CONVOLUTION LOOP DIMENSIONS AND HARDWARE DESIGN VARIABLES

	Kernel Window (width/height)	Input Feature Map (width/height)	Output Feature Map (width/height)	# of Input Feature Maps	# of Output Feature Maps
Convolution Loops	Loop-1	Loop-3	Loop-3	Loop-2	Loop-4
Convolution Dimensions (N*)	Nkx, Nky	Nix, Niy	Nox, Noy	Nif	Nof
Loop Tiling (*T)**	*Tkx, Tky*	*Tix, Tiy*	*Tox, Toy*	*Tif*	*Tof*
Loop Unrolling (*P)**	*Pkx, Pky*	*Pix, Piy*	*Pox, Poy*	*Pif*	*Pof*

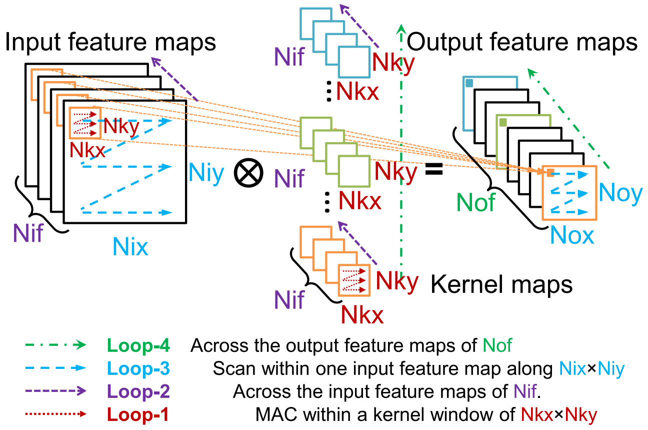

Input feature maps
Nif
Nkx
Nky
Niy
Nix
Nif
Nkx
Nky
Nif
Nkx
Nky
Output feature maps
Nof
Noy
Nox
Nif
Nkx
Nky
Kernel maps
Loop-4 Across the output feature maps of Nof
Loop-3 Scan within one input feature map along Nix×Niy
Loop-2 Across the input feature maps of Nif.
Loop-1 MAC within a kernel window of Nkx×Nky

Fig. 3. Four levels of convolution loops and their dimensions.

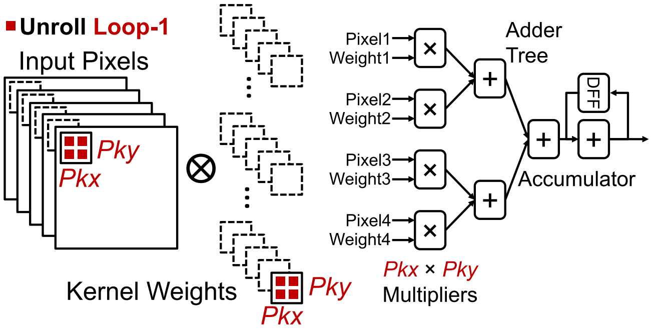


Fig. 4. Unroll loop-1 and its corresponding computing architecture.

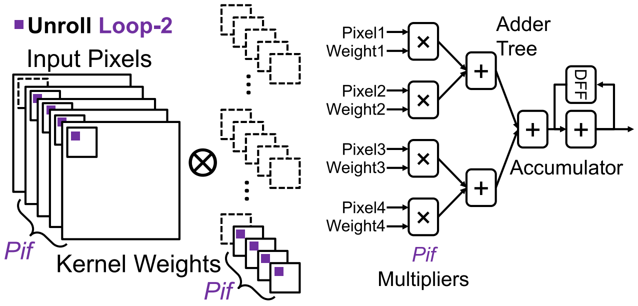


Fig. 5. Unroll loop-2 and its corresponding computing architecture.

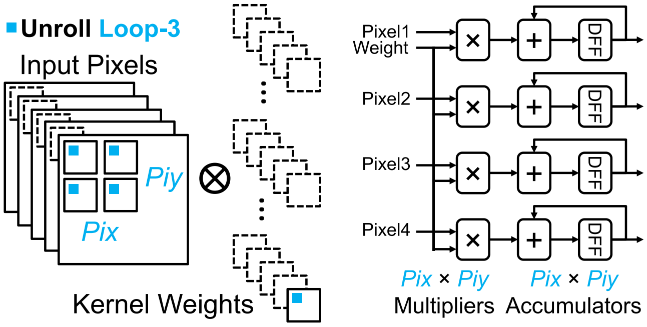


Fig. 6. Unroll loop-3 and its corresponding computing architecture.

buffers. The constraints of these dimension and variables are given by $1 \leq P* \leq T* \leq \mathrm{N}*$, where $\mathrm{N}*$, $T*$, and $P*$ denote any dimension or variable that has a prefix of capital N, T, and P, respectively. For instance, $1 \leq Pkx \leq Tkx \leq Nkx$. By default, $P*$, $T*$ and $N*$ are applied to all convolution layers.

The relationship of input and output variables is constraint by (1)–(3), where S is the stride of the sliding window and the zero padding size is included in *Nix*, *Niy*, *Tix*, and *Tiy*

$$\begin{aligned} Nix &= (Nox - 1)S + Nkx \\ Niy &= (Noy - 1)S + Nky \qquad (1) \\ Tix &= (Tox - 1)S + Nkx \\ Tiy &= (Toy - 1)S + Nky \qquad (2) \\ Pix &= Pox \\ Piy &= Poy. \qquad (3) \end{aligned}$$

1) Loop Unrolling: As illustrated in Figs. 4–7, unrolling different convolution loops leads to different parallelization of computations, which affects the optimal PE architecture with respect to data reuse opportunities and memory access patterns.

a) Loop-1 unrolling (Fig. 4): In this case, the inner product of *Pkx* × *Pky* pixels (or activations) and weights from different (x, y) locations in the same feature and kernel map are computed every cycle. This inner product requires an adder tree with fan-in of *Pkx* × *Pky* to sum the *Pkx* × *Pky* parallel multiplication results, and an accumulator to add the adder tree output with the previous partial sum.

b) Loop-2 unrolling (Fig. 5): In every cycle, *Pif* number of pixels/weights from *Pif* different feature/kernel maps at the same (x, y) location are required to compute the inner product. The inner-product operation results in the same computing structure as in unrolling Loop-1, but with a different adder tree fan-in of *Pif*.

c) Loop-3 unrolling (Fig. 6): In every cycle, *Pix* × *Piy* number of pixels from different (x, y) locations in the same feature map are multiplied with the identical weight. Hence, this weight can be reused *Pix* × *Piy* times. Since the *Pix* ×

Authorized licensed use limited to: University of Louisiana at Lafayette. Downloaded on October 19,2020 at 23:35:00 UTC from IEEE Xplore. Restrictions apply.

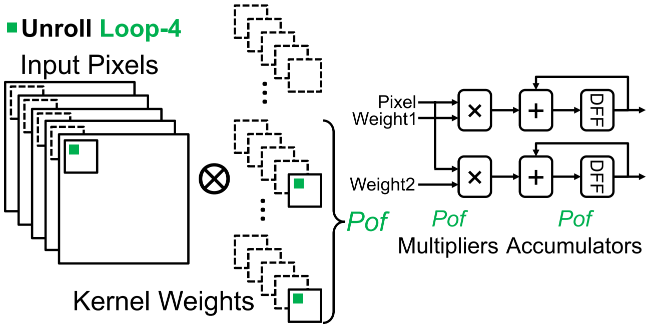


Fig. 7. Unroll loop-4 and its corresponding computing architecture.

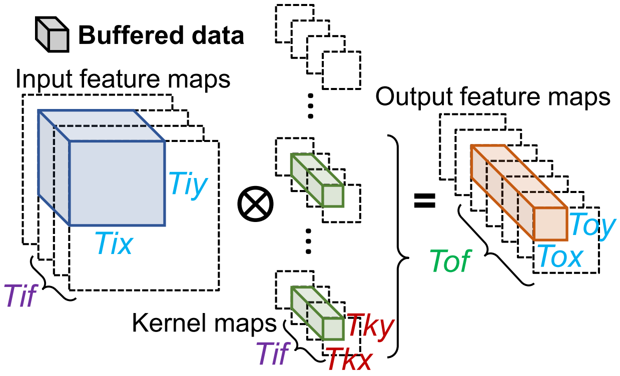


Fig. 8. Loop tiling determines the size of data stored in on-chip buffers.

Piy parallel multiplication contributes to independent *Pix* × *Piy* output pixels, *Pix* × *Piy* accumulators are used to serially accumulate the multiplier outputs and no adder tree is needed.

d) Loop-4 unrolling (Fig. 7): In every cycle, one pixel is multiplied by *Pof* weights at the same (x, y) location but from *Pof* different kernel maps, and this pixel is reused *Pof* times. The computing structure is identical to unrolling Loop-3 using *Pof* multipliers and accumulators without an adder tree.

The unrolling variable values of the four convolution loops collectively determine the total number of parallel MAC operations as well as the number of required multipliers (*Pm*)

$$Pm = Pkx \times Pky \times Pif \times Pix \times Piy \times Pof. \tag{4}$$

2) Loop Tiling: On-chip memory of FPGAs is not always large enough to store the entire data of deep CNN algorithms. Therefore, it is reasonable to use denser external DRAMs to store the weights and the intermediate pixel results of all layers.

Loop tiling is used to divide the entire data into multiple blocks, which can be accommodated in the on-chip buffers, as illustrated in Fig. 8. With proper assignments of the loop tiling size, the locality of data can be increased to reduce the number of DRAM accesses, which incurs long latency and high-power consumption. The loop tiling sets the lower bound on the required on-chip buffer size. The required size of input pixel buffer is *Tix* × *Tiy* × *Tif* × (pixel_datawidth). The size of weight buffer is *Tkx* × *Tky* × *Tif* × *Tof* × (weight_datawidth). The size of output pixel buffer is *Tox* × *Toy* × *Tof* × (pixel_datawidth).

3) Loop Interchange: Loop interchange determines the order of the sequential computation of the four convolution loops. There are two kinds of loop interchange, namely, intratile and intertile loop orders. Intratile loop order determines the pattern of data movements from on-chip buffer to PEs. Intertile loop order determines the data movement from external memory to on-chip buffer.

III. Analysis on Design Objectives of CNN Accelerator

In this section, we provide a quantitative analysis of the impact of loop design variables ($P*$ and $T*$) on the following design objectives that our CNN accelerator aims to minimize.

A. Computing Latency

The number of multiplication operations per layer (Nm) is

$$Nm = Nif \times Nkx \times Nky \times Nof \times Nox \times Noy. \tag{5}$$

Ideally, the number of computing cycles per layer should be Nm/*Pm*, where *Pm* is the number of multipliers. However, for different loop unrolling and tiling sizes, the multipliers cannot necessarily be fully utilized for every convolution dimension. The number of actual computing cycles per layer is

$$\#\_cycles = \#intertile\_cycles \times \#intratile\_cycles \tag{6}$$

where

$$\begin{aligned}\#intertile\_cycles &= \lceil Nif/Tif \rceil \lceil Nkx/Tkx \rceil \lceil Nky/Tky \rceil \\ &\quad \times \lceil Nof/Tof \rceil \lceil Nox/Tox \rceil \lceil Noy/Toy \rceil \qquad (7)\\ \#intratile\_cycles &= \lceil Tif/Pif \rceil \lceil Tkx/Pkx \rceil \lceil Tky/Pky \rceil \\ &\quad \times \lceil Tof/Pof \rceil \lceil Tox/Pox \rceil \lceil Toy/Poy \rceil. \qquad (8)\end{aligned}$$

Here, we assume that the multipliers receive input data continuously without idle cycles. If the ratio of $N*$ to $T*$ or $T*$ to $P*$ is not an integer, the multipliers or the external memory transactions are not fully utilized. In addition to considering computing latency, memory transfer delay must also be considered for the overall system latency.

B. Partial Sum Storage

A partial sum (psum) is the intermediate result of the inner-product operation that needs to be accumulated over several cycles to obtain one final output data. Therefore, partial sums need to be stored in memory for the next few cycles and sometimes have to be moved between PEs. An efficient acceleration strategy has to minimize the number of partial sums and process them locally as soon as possible to reduce data movements.

The flowchart to calculate the number of partial sums stored in memory (#psum) is shown in Fig. 9. To obtain one final output pixel, we need to finish Loop-1 and Loop-2. Therefore, if both Loop-1 and Loop-2 are fully unrolled, the final output pixel can be obtained right after the inner-product operations with minimal #psum. If the loop tile size can cover all pixels and weights in Loop-1 (*Tkx* = *Nkx* and *Tky* = *Nky*) and Loop-2 (*Tif* = *Nif*), then the partial sums can be consumed

Authorized licensed use limited to: University of Louisiana at Lafayette. Downloaded on October 19,2020 at 23:35:00 UTC from IEEE Xplore. Restrictions apply.

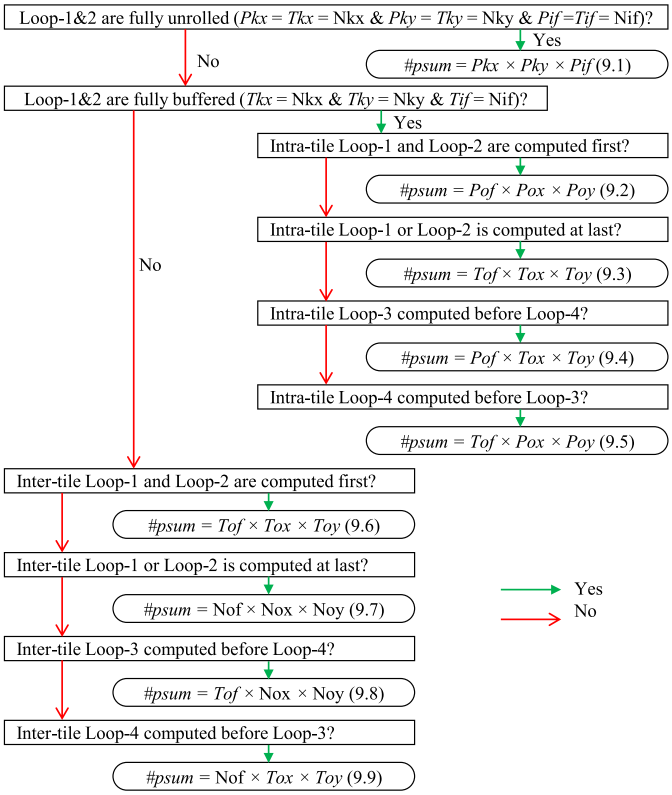

Loop-1&2 are fully unrolled (Pkx = Tkx = Nkx & Pky = Tky = Nky & Pif =Tif = Nif)?
Yes
No
#psum = Pkx × Pky × Pif (9.1)
Loop-1&2 are fully buffered (Tkx = Nkx & Tky = Nky & Tif = Nif)?
Yes
No
Intra-tile Loop-1 and Loop-2 are computed first?
#psum = Pof × Pox × Poy (9.2)
Intra-tile Loop-1 or Loop-2 is computed at last?
#psum = Tof × Tox × Toy (9.3)
Intra-tile Loop-3 computed before Loop-4?
#psum = Pof × Tox × Toy (9.4)
Intra-tile Loop-4 computed before Loop-3?
#psum = Tof × Pox × Poy (9.5)
Inter-tile Loop-1 and Loop-2 are computed first?
#psum = Tof × Tox × Toy (9.6)
Inter-tile Loop-1 or Loop-2 is computed at last?
#psum = Nof × Nox × Noy (9.7)
Inter-tile Loop-3 computed before Loop-4?
#psum = Tof × Nox × Noy (9.8)
Inter-tile Loop-4 computed before Loop-3?
#psum = Nof × Tox × Toy (9.9)
Yes
No

Fig. 9. Design space exploration of the total number of partial sums that need to be stored in memory [15].

within this tile as described in (9.2)–(9.5) inside Fig. 9. In this case, the number of partial sums, determined by $P*$ or $T*$, is small and can be stored in local registers [(9.2) inside Fig. 9] or in on-chip buffers [(9.3) inside Fig. 9]. If the loop tile cannot include all data for Loop-1 and Loop-2, partial sums from one tile need to be stored in on-chip or off-chip memory until it is consumed by another tile as in (9.6)–(9.9) inside Fig. 9. In this case, the partial sums need to be stored in on-chip buffers [(9.6) inside Fig. 9] or even in external memory [(9.7) inside Fig. 9]. The loop computing order also affects the number of partial sums, and the earlier Loop-1 and Loop-2 are computed, the fewer is the number of partial sums. The requirement to store partial sums in different levels of memory hierarchy significantly worsens data movements and associated energy cost [8] since partial sums involve both read and write memory operations and typically require higher precision than pixels and weights.

C. Data Reuse

Reusing pixels and weights reduces the number of read operations of on-chip buffers. There are mainly two types of data reuse: spatial reuse and temporal reuse. Spatial reuse means that, after reading data from on-chip buffers, a single pixel or weight is used for multiple parallel multipliers within one clock cycle. On the other hand, temporal reuse means that a single pixel or weight is used for multiple consecutive cycles.

Having Pm parallel multiplications per cycle requires Pm pixels and Pm weights to be fed into the multipliers. The number of distinct weights required per cycle is

$$Pwt = Pof \times Pif \times Pkx \times Pky. \tag{9}$$

If Loop-1 is not unrolled ($Pkx = 1$, $Pky = 1$), the number of distinct pixels required per cycle (Ppx) is

$$Ppx = Pif \times Pix \times Piy. \tag{10}$$

Otherwise, Ppx is

$$Ppx = Pif \times ((Pix - 1)S + Pkx) \times ((Piy - 1)S + Pky). \tag{11}$$

Note that "distinct" only means that the pixels/weights are from different feature/kernel map locations and their values may be the same. The number of times a weight is spatially reused in one cycle is

$$Reuse\_wt = Pm/Pwt = Pix \times Piy \tag{12}$$

where the spatial reuse of weights is realized by unrolling Loop-3 ($Pix > 1$ or $Piy > 1$). The number of times of a pixel is spatially reused in one cycle (Reuse_px) is

$$Reuse\_px = Pm/Ppx. \tag{13}$$

If Loop-1 is not unrolled, Reuse_px is

$$Reuse\_px = Pof \tag{14}$$

otherwise, Reuse_px is

$$Reuse_{px} = \frac{Pof \times Pkx \times Pky \times Pix \times Piy}{((Pix - 1)S + Pkx) \times ((Piy - 1)S + Pky)}. \tag{15}$$

The spatial reuse of pixels is realized by either unrolling Loop-4 ($Pof > 1$) or unrolling both Loop-1 and Loop-3 together. Only unrolling Loop-1 ($Pix = 1$, $Piy = 1$) or only unrolling Loop-3 ($Pkx = 1$, $Pky = 1$) hampers reusing pixels, and Reuse_$px = Pof$.

If intra-tile Loop-3 is computed first, the weights can be reused for $Tox \times Toy/(Pox \times Poy)$ consecutive cycles. If intra-tile Loop-4 is computed first, the pixels can be reused for Tof/Pof consecutive cycles.

D. Access of On-Chip Buffer

With the data reuse, the number of on-chip buffer accesses can be significantly reduced. Without any data reuse, the total read operations from on-chip buffers for both pixels and weights are Nm, as every multiplication needs one pixel and one weight. With data reuse, the total number of read operations from on-chip buffers for weights becomes

$$\#read\_wt = Nm/Reuse\_wt \tag{16}$$

and the total number of read operations for pixels is

$$\#read\_px = Nm/Reuse\_px. \tag{17}$$

If the final output pixels cannot be obtained within one tile, their partial sums are stored in buffers. The number of write and read operations to/from buffers for partial sums per cycle is $2 \times Pof \times Pox \times Poy$, where all partial sums

Authorized licensed use limited to: University of Louisiana at Lafayette. Downloaded on October 19,2020 at 23:35:00 UTC from IEEE Xplore. Restrictions apply.

generated by Loop-1 (*Pkx*, *Pky*) and Loop-2 (*Pif*) are already summed together right after multiplications. The total number of writes/reads to/from buffers for partial sums is

$$\#wr\_rd\_psum = \#\_cycles \times (2 \times Pof \times Pox \times Poy). \tag{18}$$

The number of times output pixels are written to on-chip buffers (i.e., *#write_px*) is identical to the total number of output pixels in the given CNN model. Finally, the total number of on-chip buffer accesses is

$$\#buffer\_access = \#read\_px + \#read\_wt + \#wr\_rd\_psum + \#write\_px. \tag{19}$$

E. Access of External Memory

In our analysis, both the weights and intermediate results of pixels are assumed to be stored in external memory (DRAM), which is a necessity when mapping large-scale CNNs on moderate FPGAs. The costs of DRAM accesses are higher latency and energy than on-chip block RAM (BRAM) memory accesses [8], [20], and therefore it is important to reduce the number of external memory accesses to improve the overall performance and energy efficiency. The minimum number of DRAM accesses is achieved by having sufficiently large on-chip buffers and proper loop computing orders, such that every pixel and weight needs to be transferred from DRAM only once. Otherwise, the same pixel or weight has to be read multiple times from DRAM to be consumed for multiple tiles.

The flowchart to estimate the number of DRAM accesses is shown in Fig. 10, where #DRAM_*px* and #DRAM_*wt* denote the number of DRAM access of one input pixel and one weight, respectively. After fetched out of DRAM, all data should be exhaustedly utilized before being kicked out of the buffer. Therefore, if the tile size or the on-chip buffer can fully cover either all input pixels or all weights of one layer, the minimum DRAM access can be achieved as (10.8) inside Fig. 10. By computing Loop-3 first, weights stored in buffer are reused and #DRAM_*wt* is reduced as in (10.1) and (10.5) inside Fig. 10. Similarly, by computing Loop-4 first, pixels can be reused to reduce #DRAM_*px* as in (10.3) and (10.6) inside Fig. 10. However, computing Loop-3 or Loop-4 first may postpone the computation of Loop-1 or Loop-2, which would lead to a large number of partial sums.

IV. Loop Optimization in Related Works

In this section, the acceleration schemes of the state-of-the-art hardware CNN accelerators are compared. The loop unrolling strategy of current designs can be categorized into the four types:

1) [Type-(A)] unroll loop-1, loop-2, loop-4 [11], [13], [17], [19];
2) [Type-(B)] unroll loop-2, loop-4 [9], [14];
3) [Type-(C)] unroll loop-1, loop-3 [7], [8], [21];
4) [Type-(D)] unroll loop-3, loop-4 [15], [16], [18].

By unrolling Loop-1, Loop-2, and Loop-4 in type-(A), parallelism is employed in kernel maps, input and output feature maps. However, kernel size (*Nkx* × *Nky*) is normally very small (≤11 × 11) so that it cannot provide sufficient parallelism and other loops need to be further unrolled. A more challenging problem is that kernel size may vary considerably across different convolution layers in a given CNN model (e.g., AlexNet [2], ResNet [5]), which may cause workload imbalance and inefficient utilization of the PEs [21]. To address this, PEs need to be configured differently for layers with different kernel sizes [10], which increase control complexity.

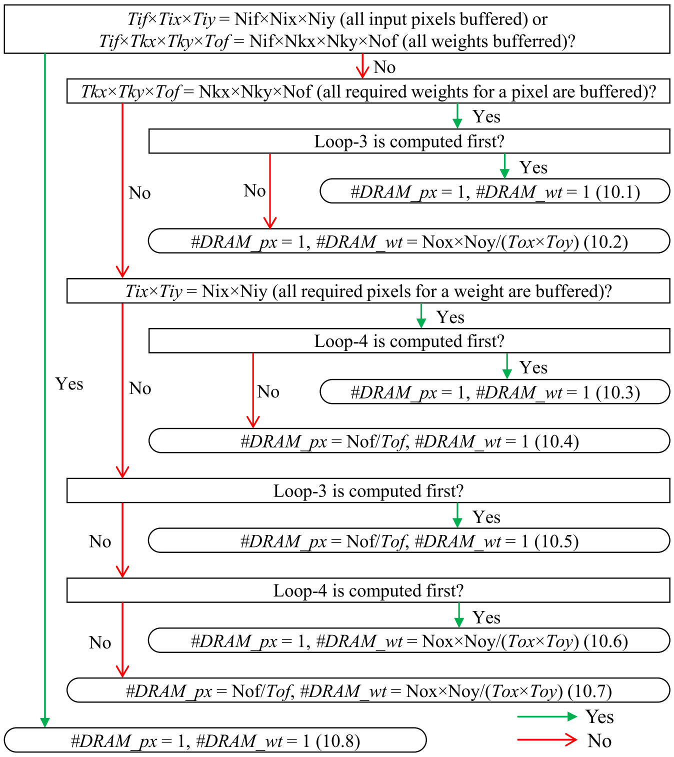

Tif×*Tix*×*Tiy* = Nif×Nix×Niy (all input pixels buffered) or
Tif×*Tkx*×*Tky*×*Tof* = Nif×Nkx×Nky×Nof (all weights bufferred)?
No
Tkx×*Tky*×*Tof* = Nkx×Nky×Nof (all required weights for a pixel are buffered)?
Yes
Loop-3 is computed first?
Yes
No
No
#DRAM_px = 1, *#DRAM_wt* = 1 (10.1)
#DRAM_px = 1, *#DRAM_wt* = Nox×Noy/(*Tox*×*Toy*) (10.2)
Tix×*Tiy* = Nix×Niy (all required pixels for a weight are buffered)?
Yes
Loop-4 is computed first?
Yes
Yes
No
No
#DRAM_px = 1, *#DRAM_wt* = 1 (10.3)
#DRAM_px = Nof/*Tof*, *#DRAM_wt* = 1 (10.4)
Loop-3 is computed first?
Yes
No
#DRAM_px = Nof/*Tof*, *#DRAM_wt* = 1 (10.5)
Loop-4 is computed first?
Yes
No
#DRAM_px = 1, *#DRAM_wt* = Nox×Noy/(*Tox*×*Toy*) (10.6)
#DRAM_px = Nof/*Tof*, *#DRAM_wt* = Nox×Noy/(*Tox*×*Toy*) (10.7)
#DRAM_px = 1, *#DRAM_wt* = 1 (10.8)
Yes
No

Fig. 10. Design space exploration of the number of external memory accesses.

In type-(C), every row in the kernel window is fully unrolled (*Pkx* = *Nkx*) and Loop-3 is also partially unrolled. By this means, pixels can be reused by the overlapping caused by Loop-1 and Loop-3 as in (15), and weight reuse can also be realized by unrolling Loop-3 as in (12). However, Loop-4 is not unrolled and further pixel reuse cannot be achieved. The PE efficiency issue caused by unrolling Loop-1 also affects type-(C) [21].

In type-(A) and type-(B), Loop-3 is not unrolled, which implies that weights cannot be reused. Type-(B) only unrolls Loop-2 and Loop-4, but *Nif* × *Nof* of the first convolution layer is usually small (≤3 × 96) and cannot provide sufficient parallelism, which results in low utilization and throughput. If the first layer is computation bounded or the DRAM delay is not overlapped with the computation, the throughput degradation will affect the overall performance, especially for shallow CNNs, e.g., AlexNet and NiN.

In type-(D), both Loop-3 and Loop-4 are unrolled so that both pixels and weights can be reused. In addition, *Nox* × *Noy* × *Nof* (≥7 × 7 × 64) is very large across all the convolution layers in AlexNet, VGG, and ResNet so that high

Authorized licensed use limited to: University of Louisiana at Lafayette. Downloaded on October 19,2020 at 23:35:00 UTC from IEEE Xplore. Restrictions apply.

level of parallelism can be achieved even for largest FPGA available with ∼3600 DSP slices. By this means, a uniform configuration and structure of PEs can be applied for all the convolution layers.

Loop tiling has been used in prior hardware CNN accelerators to fit the large-scale CNN models into limited on-chip buffers. However, only a few prior works [13], [18] have shown their tiling configurations that determine the on-chip buffer size, but the tradeoff between the loop tiling size and the number of external memory accesses is not explored.

The impact of loop interchange has not been rigorously studied in prior works, but it can greatly impact the number of partial sums as well as the resulting data movements and memory access.

V. Proposed Acceleration Scheme

The optimization process of our proposed acceleration scheme is presented in this section, which includes appropriate selection of the convolution loop design variables.

A. Minimizing Computing Latency

We set variables $P*$ to be the common factors of $T*$ for all the convolution layers to fully utilize PEs, and $T*$ to be the common factors of N$*$ to make full use of external memory transactions. For CNN models with only small common factors, it is recommended to set $\lceil \mathrm{N}*/T* \rceil$ − N$*$/$T*$ and $\lceil T*/P* \rceil$ − $T*/P*$ as small as possible to minimize the inefficiency caused by the difference in sizes of CNN models.

B. Minimizing Partial Sum Storage

To reduce the number and movements of partial sums, both Loop-1 and Loop-2 should be computed as early as possible or unrolled as much as possible. To avoid the drawback of unrolling Loop-1 as discussed in Section IV and maximize the data reuse as discussed in Section III-C, we decide to unroll Loop-3 ($Pox > 1$ or $Poy > 1$) and Loop-4 ($Pof > 1$). By this means, we cannot attain the minimum partial sum storage, as (9.1) inside Fig. 9.

Constrained by $1 \le P* \le T* \le \mathrm{N}*$, the second least number of partial sum storage is achieved by (9.2) among (9.2)–(9.9) inside Fig. 9. To satisfy the condition for (9.2), we serially compute Loop-1 and Loop-2 first and ensure the required data of Loop-1 and Loop-2 are buffered, i.e., $Tkx = Nkx$, $Tky = Nky$ and $Tif = Nif$. Therefore, we only need to store $Pof \times Pox \times Poy$ number of partial sums, which can be retained in local registers with minimum data movements.

C. Minimizing Access of On-Chip Buffer

The number of on-chip buffer accesses is minimized by unrolling Loop-3 to reuse weights as shown in (12) and unrolling Loop-4 to reuse pixels as shown in (14). As our partial sums are kept on local registers, they do not add overhead to the buffer access and storage.

D. Minimizing Access of External Memory

As we first compute Loop-1 and Loop-2 to reduce partial sums, we cannot achieve the minimum number of DRAM access described in (10.1) and (10.3) inside Fig. 10, where neither the pixels nor the weights are fully buffered for one convolution layer. Therefore, we can only attain the minimum DRAM access by assigning sufficient buffer size for either all pixels or all weights of each layer as in (10.8) inside Fig. 10.

Then, the optimization of minimizing the on-chip buffer size while having minimum DRAM access is formulated as

$$\begin{aligned} &\mathbf{min}\ bits\_BUF\_px\_wt \\ &\mathbf{s.t.}\ \#Tile\_px_L = 1or\#Tile\_wt_L = 1 \\ &\quad\mathbf{with}\ \ \forall L \in\ [1, \#CONVs] \end{aligned} \tag{20}$$

where #Tile_px_L and #Tile_wt_L denote the number of tiling blocks for input pixels and weights of layer L, respectively, and #CONVs is the number of convolution layers. bits_BUF_px_wt is the sum of pixel buffer size (bits_BUF_px) and weight buffer size (bits_BUF_wt), which are given by

$$bits\_BUF\_px\_wt = bits\_BUF\_px + bits\_BUF\_wt. \tag{21}$$

Both pixel and weight buffers need to be large enough to cover the data in one tiling block for all the convolution layers. This is expressed as

$$\begin{aligned} &bits\_BUF\_px \\ &\quad = MAX(words\_px_L) \\ &\qquad \times pixel\_datawidth\ with\ L\ \in\ [1, \#CONVs] \end{aligned} \tag{22}$$

$$\begin{aligned} &bits\_BUF\_wt \\ &\quad = MAX(words\_wt_L) \\ &\qquad \times weight\_datawidth\ with\ L \in\ [1, \#CONVs] \end{aligned} \tag{23}$$

where words_px_L and words_wt_L denote the number of pixels and weights of one tiling block in layer L, respectively. These are expressed in terms of loop tiling variables as follows:

$$words\_px_L = Tix_L \times Tiy_L \times Tif_L + Tox_L \times Toy_L \times Tof_L \tag{24}$$

$$words\_wt_L = Tof_L \times Tif_L \times Tkx_L \times Tky_L \tag{25}$$

where words_px_L is comprised of both input and output pixels. The number of tiles in (20) is also determined by $T*$ variables

$$\#Tile\_px_L = \lceil Nif_L/Tif_L \rceil \times \lceil Nox_L/Tox_L \rceil \times \lceil Noy_L/Toy_L \rceil \tag{26}$$

$$\begin{aligned} \#Tile\_wt_L = &\lceil Nkx_L/Tkx_L \rceil \times \lceil Nky_L/Tky_L \rceil \times \lceil Nif_L/Tif_L \rceil \\ &\times \lceil Nof_L/Tof_L \rceil. \end{aligned} \tag{27}$$

By solving (20), we can find an optimal configuration of $T*$ variables that result in minimum DRAM access and on-chip buffer size. However, since we have already set $Tkx = Nkx$, $Tky = Nky$, $Tif = Nif$ as in Section V-B, we can only achieve a suboptimal solution by tuning Tox, Toy and Tof, resulting in larger buffer size requirement. If the available on-chip memory is sufficient, we set $Tox = Nox$ so that an entire row can be buffered to benefit the direct memory access (DMA) transactions with continuous data.

Authorized licensed use limited to: University of Louisiana at Lafayette. Downloaded on October 19,2020 at 23:35:00 UTC from IEEE Xplore. Restrictions apply.

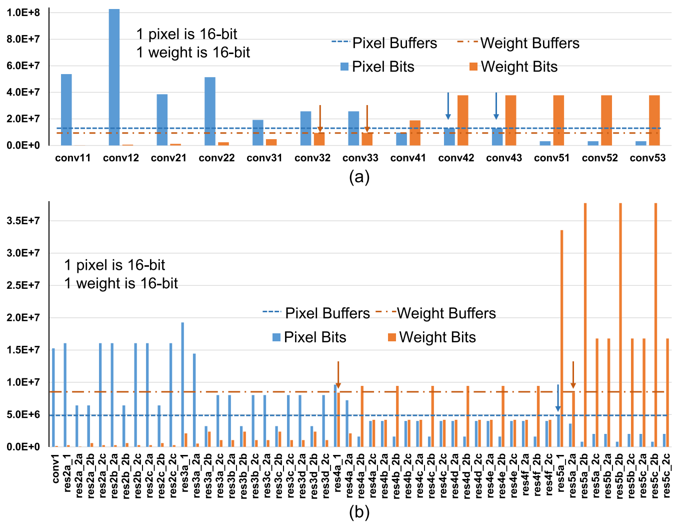


Fig. 11. To guarantee minimum DRAM accesses, either all pixels (blue bars) are covered by pixel buffers (blue dashed lines) or all weights are covered by weight buffers in one layer. Then, we try to lower the total buffer sizes/lines. (a) Pixels and weights distribution of convolution layers in VGG-16. (b) Pixels and weights distribution of convolution layers in ResNet-50.

Finally, we have to solve (20) by searching *Toy* and *Tof*, because it has a nonlinear objective function and constraints with integer variables. Since *Toy* and *Tof* in VGG-16 consist of 2 × #CONVs = 26 variables and each variable can have about four candidate values constrained by $T*/P*$ = integer and N^*/T^* = integer, the total number of *Toy* and *Tof* configurations is about $4^{26} = 4.5 \times 10^{15}$, which becomes an enormous solution space. In ResNet-50/ResNet-152, the #CONVs are increased to be 53 and 155, respectively, which makes the solution space even larger to be about $4^{106} = 6.6 \times 10^{63}$ and $4^{310} = 4.4 \times 10^{186}$, respectively. Therefore, it is impossible to enumerate all the candidate solutions.

In this paper, we propose to empirically find a satisfactory solution for a given on-chip memory capacity that takes advantage of the property of CNNs. CNNs normally have large pixel data volume and small weight sizes in the beginning few layers. As we proceed into deeper layers, the pixel sizes become smaller with extracted features, and the weight sizes become larger with more channels. This trend is illustrated in Fig. 11, where the bars denote data sizes in each convolution layer. To benefit from the data distribution property in different layers, we only need to make pixel buffers fully cover the last few layers and weight buffers fully cover the beginning few layers. Then, the middle layers with both relatively large pixel and weight sizes become the constraints of the buffer sizes, and we only need to take care of these bounding layers, which significantly shrinks the solution space. The dashed lines in Fig. 11 are the minimal buffer sizes we found while guaranteeing minimum DRAM accesses, and the bounding layers are pointed out by arrows. If this buffer size still cannot be fit into the FPGA on-chip memory, then we need to either change the tiling strategy or decrease the buffer sizes at the cost of more DRAM accesses as discussed in [15].

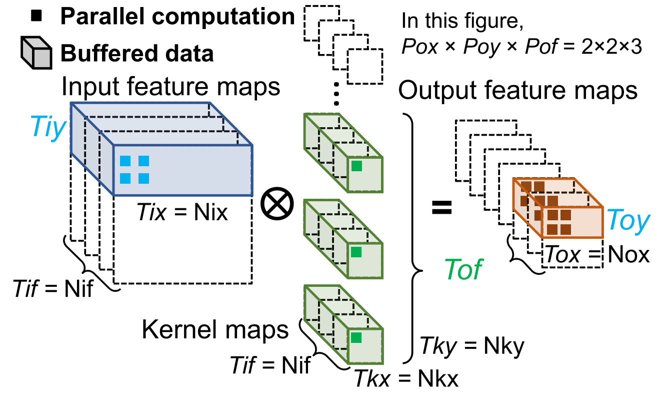


Fig. 12. Optimized loop unrolling and tiling strategy. The parallelism is within one feature map (*Pox* × *Poy*) and across multiple kernels (*Pof*). The tiling variables *Tiy*, *Toy*, and *Tof* can be tuned that decide the buffer sizes.

E. Optimized Loop Design Variables

According to the aforementioned optimization process, we propose a convolution acceleration scheme for a high-performance and low-communication CNN accelerator, which is visualized in Fig. 12.

1) Loop Unrolling: For all the convolution layers, Loop-1 and Loop-2 are not unrolled, which means $Pkx = 1$, $Pky = 1$ and $Pif = 1$. According to (7) and (8), *Pox*, *Poy* and *Pof* are set to be the common factors of the feature maps (*Nox*, *Noy*) and output channels (*Nof*), respectively, to fully utilize the multipliers. The configurations of *Pox*, *Poy,* and *Pof* of different CNNs on different FPGAs are listed in Table II, which are largely constrained by the available computing resources. By setting $P*$ to be constant across all the convolution layers, a uniform structure and mapping of PEs can be realized to reduce the architecture complexity.

2) Loop Tiling: For loop tiling, we set $Tkx = Nkx$, $Tky = Nky$, $Tif = Nif$ as described in Section V-B and shown in Fig. 12 so that data used in Loop-1 and Loop-2 are all buffered and $Tox = Nox$ to benefit DMA transfer. Details of *Toy* and *Tof* are described in Section V-D.

3) Loop Interchange: For loop interchange, we first serially compute Loop-1 and then Loop-2 as described in Section V-B. Finally, we compute Loop-3 and Loop-4, where the exact computation order of these two loops does not have a pronounced impact on the cost, based on our $P*$ and $T*$ choices.

VI. Proposed CNN Accelerator

To implement the optimized convolution acceleration scheme in Section V-E, a data router is proposed with high flexibility for different convolution sliding settings, e.g., strides and zero paddings, using variant data buses. A corresponding hardware PE architecture is also designed that minimizes on/off-chip memory accesses and data movements.

A. Data Bus From Buffer to PE (BUF2PE)

In [15] and [16], a register array architecture is designed to rearrange and direct the pixel stream from buffers into PEs. This method takes advantage of convolution stride being

Authorized licensed use limited to: University of Louisiana at Lafayette. Downloaded on October 19,2020 at 23:35:00 UTC from IEEE Xplore. Restrictions apply.

TABLE II
OUR IMPLEMENTATION OF DIFFERENT CNNS ON DIFFERENT FPGAS

FPGA / Technology	Intel Stratix V GXA7 / 28 nm				Intel Arria 10 GX 1150 / 20 nm			
CNN Model	NiN	VGG-16	ResNet-50	ResNet-152	NiN	VGG-16	ResNet-50	ResNet-152
Clock (MHz)	150	150	150	150	200	200	200	200
# of Operations (GOP)	2.20	30.95	7.74	22.62	2.20	30.95	7.74	22.62
# of Parameters	7.59 M	138.3 M	25.5 M	60.4 M	7.59 M	138.3 M	25.5 M	60.4 M
Precision (fixed)	16 bit	16 bit	16 bit	16 bit	16 bit	16 bit	16 bit	16 bit
$Pox \times Poy \times Pof$	7 × 7 × 32	7 × 7 × 32	7 × 7 × 24	7 × 7 × 24	7 × 7 × 64	7 × 7 × 64	7 × 7 × 64	7 × 7 × 64
# of MAC Units	1,568	1,568	1,176	1,176	3,136	3,136	3,136	3,136
DSP Utilization	256 (100%)	256 (100%)	256 (100%)	256 (100%)	1,518 (100%)	1,518 (100%)	1,518 (100%)	1,518 (100%)
Logic Element (ALMs)	228K (97%)	218K (93%)	176K (75%)	185K (78%)	161K (38%)	138K (32%)	221K (52%)	235K (55%)
On-chip BRAM (M20K)	1,512 (59%)	2,210 (86%)	1,950 (76%)	2,385 (93%)	1,528 (56%)	2,232 (82%)	1,931 (71%)	2,365 (87%)
Latency/Image (ms)	7.9	88.8	31.82	81.8	3.8	43.2	12.7	32.0
Overall Throughput (GOPS)	278.2	348.8	243.3	276.6	584.8	715.9	611.4	707.2

Throughput (GOPS) = # Operations (GOP) / Latency per image (s)

1 in VGG-16 so that pixels can be reused by the adjacent register array in the next computing cycles. However, if stride is 2 or more, which frequently occurs in CNN algorithms [2], [3], [5], pixels need to wait for Nkx × (Stride-1) cycles to be reused by the neighboring register array. This makes the control logic and wire routing among registers much more complicated. Therefore, we propose a BUF2PE data bus in Fig. 13 to implement the dataflow using FIFO to temporally store pixels to be reused by the adjacent register array. This method is similar to line buffer design in [22], where FIFOs are used to align pixels from multiple feature rows to a kernel window so that parallelism can be employed within a kernel window, i.e., unrolling Loop-1, whereas this paper unrolls Loop-3 to parallel compute within one feature map. By this means, the wire routing within and across register arrays is simplified, and the data router can follow the same pattern for convolution with different strides and zero paddings to improve the accelerator flexibility.

The detailed design of BUF2PE data bus is illustrated in Fig. 13. Pixels from input buffers are loaded into the corresponding registers as shown by the blue dashed box to the blue solid box. Then, the pixels are sent to PEs or MAC units and are also sent to FIFOs during cycles 0 to 5, waiting to be reused by the adjacent register array. Register arrays except the rightmost one start reading input pixels from FIFOs at cycle 3, as shown by the purple pixels in Fig. 13. Meanwhile, the new pixels are fed into the rightmost register array from buffers. In this paper, the offset caused by west zero padding is handled by shifting the connection between buffers and register arrays, whereas [15] has to change the storage pattern within one address of input buffer by a padding offset that increases the complexity of transferring data from DRAM to buffers.

The coarse-grained dataflow is shown in Fig. 14 at feature map row level for stride = 1 and stride = 2. The data flow in Fig. 14(a) is the same as Fig. 13, where more clock cycles of operation is shown after cycle 8. In Fig. 14(b), the dataflow with stride = 2 and zero padding = 3 is shown, which follows the same pattern as the case with stride = 1. The buffer storage pattern is adjusted according to different stride and padding settings. Three rows of zeros are added to the buffer due to the north zero padding of 3. With stride = 2, every two rows of pixels are continuously distributed across Poy buffer banks. These adjustments are handled by the buffer write enable and address signals during the reception of pixels from DRAM. Since the data movement within a register array or a

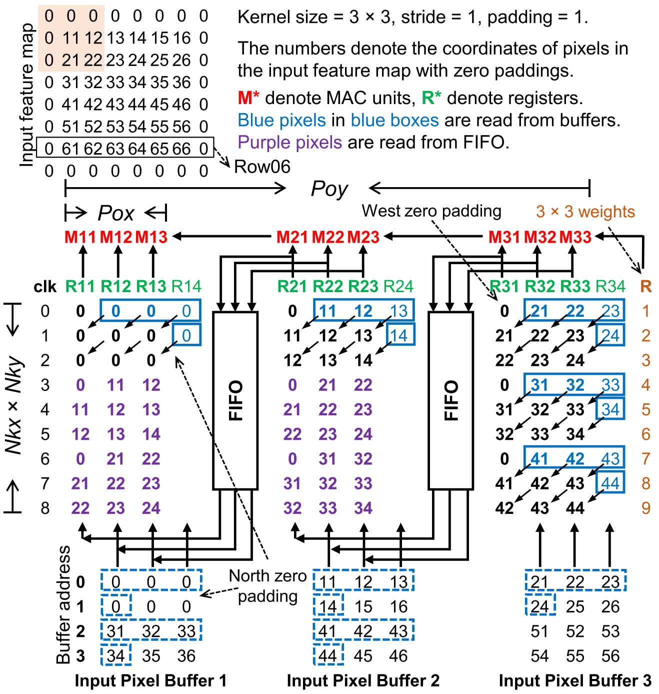
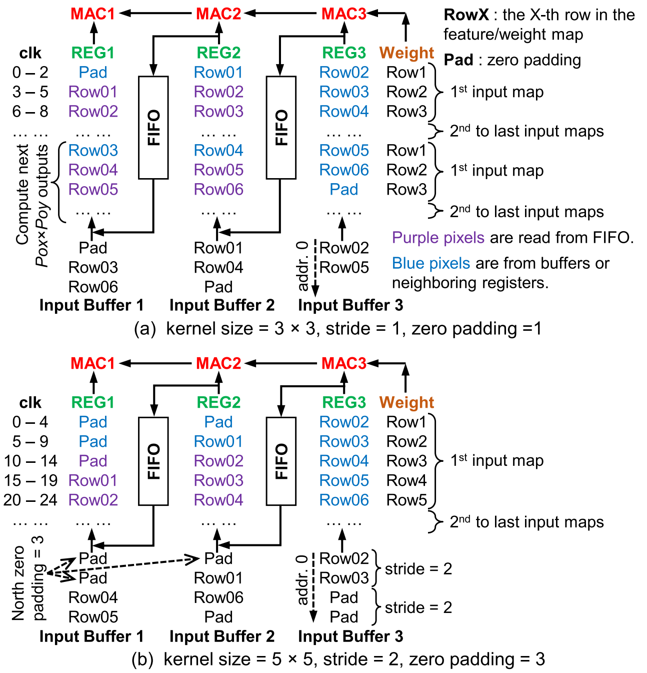

Kernel size = 3 × 3, stride = 1, padding = 1.
The numbers denote the coordinates of pixels in the input feature map with zero paddings.
M* denote MAC units, R* denote registers.
Blue pixels in blue boxes are read from buffers.
Purple pixels are read from FIFO.
Input feature map
Row06
Pox
Poy
West zero padding
3 × 3 weights
M11 M12 M13
M21 M22 M23
M31 M32 M33
clk
R11 R12 R13 R14
R21 R22 R23 R24
R31 R32 R33 R34
R
Nkx × Nky
FIFO
FIFO
North zero padding
Buffer address
Input Pixel Buffer 1
Input Pixel Buffer 2
Input Pixel Buffer 3

Fig. 13. BUF2PE data bus directs the convolution pixel dataflow from input buffers to PEs (i.e., MAC units), where $Pox = 3$ and $Poy = 3$.



(a) kernel size = 3 × 3, stride = 1, zero padding =1

(b) kernel size = 5 × 5, stride = 2, zero padding = 3

Fig. 14. Coarse-grained designs of BUF2PE data buses for (a) strides = 1 and zero padding = 1 and (b) stride = 2 and zero padding = 3.

feature map row is different for different settings of stride and zero padding, various BUF2PE data buses are needed for each dataflow, and the set of data buses are called data router. If these settings are identical, one BUF2PE bus can handle different kernel sizes (*Nkx* × *Nky*) without penalty of idle cycles as we serially compute Loop-1. Therefore, the BUF2PE bus in Fig. 14(b) can be applied for conv1 in ResNet with stride = 2 and zero padding = 3. For other sliding settings in ResNet, e.g. stride = 2 and zero padding = 0, the corresponding variants of BUF2PE buses are designed to direct the dataflow. The global control logic controls the switch among different BUF2PE buses inside the data router.

After *Nkx* × *Nky* cycles, we complete one kernel window sliding (Loop-1) and move to the next input feature map with the same dataflow until the last one as shown in Fig. 14. After *Nkx* × *Nky* ×*Nif* cycles, both Loop-1 and Loop-2 are completed and we obtain *Pox* × *Poy* × *Pof* final output pixels.

In summary, the proposed dataflow is scalable to *Nkx* × *Nky* by changing the control logic, and it can handle various sliding settings using variant BUF2PE data buses inside the data router, where the MAC units are reused and kept busy.

B. Convolution PE Architecture

The PE architecture of convolution layers shown in Fig. 15 is designed according to the proposed acceleration strategy and dataflow. It is comprised of *Pox* × *Poy* × *Pof* PEs, and every PE in our architecture is an independent MAC unit consisting of one multiplier followed by an accumulator. As Loop-1 and Loop-2 are not unrolled, no adder tree is needed to sum the multiplier outputs. The partial sum is consumed inside each MAC unit until the final results are obtained, such that the data movements of partial sums are minimized. Pixels read from input pixel buffers are shared by *Pof* MAC units and sliding overlapped pixels are also reused by the data router. Weights read from weight buffers are shared by *Pox* × *Poy* MAC units. The proposed architecture is implemented with parameterized Verilog codes and is highly scalable to different CNN models in FPGAs or even ASICs by modifying design variables such as *Pox*, *Poy* and *Pof*.

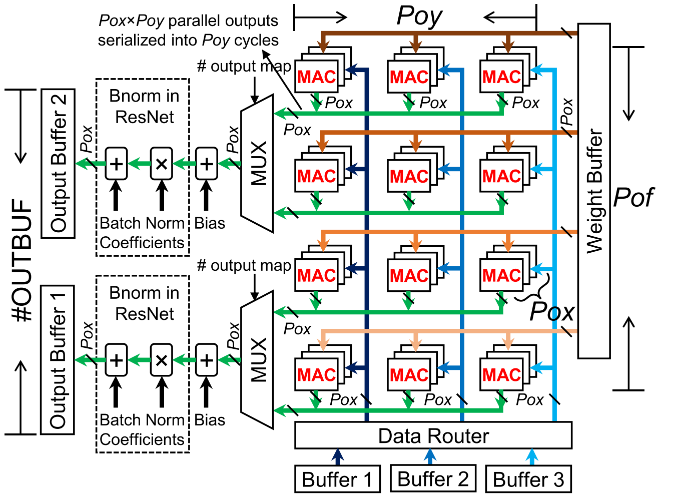


Fig. 15. Convolution acceleration architecture with *Pox* × *Poy* × *Pof* MAC units.

After the completion of Loop-1 and Loop-2, the partial sums need to be added with biases as in Fig. 1 to obtain the final output pixels. Therefore, every *Nkx* × *Nky* × *Nif* cycles, MAC units output the partial sums into the adders to add with biases. Since *Poy* < *Nkx* × *Nky* × *Nif* for all the layers, we serialize the *Pox* × *Poy* × *Pof* MAC outputs into *Poy* cycles. Then, we only need *Pox* × *Pof* adders to add the biases in parallel. The data width of one output buffer can also be reduced to be *Pox*, and we store the pixels of one output feature map in one buffer bank, which could need totally *Pof* output buffers. If *Pof* is large, e.g., *Pof* = 64, it would require many output buffers with shallow depth, resulting in low utilization of on-chip BRAMs (e.g., M20K memory block). In addition, batch normalization (Bnorm) layers in ResNet still need *Pox* × *Pof* adders and multipliers that are expensive. We further serialize the *Pox* × *Pof* parallel outputs to be *Pox* × #OUTBUF using multiplexers with neighboring output feature maps stacked in one output buffer, as illustrated in Fig. 15. In ResNet, we set #OUTBUF = 16 to ensure *Poy* × *Pof* / (#OUTBUF) < *Nkx* × *Nky* × *Nif* or the number of serial output cycles is smaller than the MAC unit output interval cycles. By this means, the parallelism of adders and multipliers for bias and Bnorm is significantly reduced, as well as the output buffer bandwidth and the used M20K BRAMs.

C. Pooling Layers

Pooling is commonly used to reduce the feature map dimension by replacing pixels within a kernel window (e.g., 2 × 2, 3 × 3) by their maximum or average value. The output pixels from previous convolution layers are stored row-by-row in the output pixel buffers. As pooling operation only need pixels,

Authorized licensed use limited to: University of Louisiana at Lafayette. Downloaded on October 19,2020 at 23:35:00 UTC from IEEE Xplore. Restrictions apply.

after one tile of convolution is finished, we directly compute pooling with pixels read from output pixel buffers to eliminate the access of external memory. The unrolling factors of all the pooling layers are the same. Since the width of output pixel buffer is *Pox*, we can enable *Pox* × #OUTBUF parallel pooling operations, which is large enough considering that pooling layers involve much less operations compared to convolution layers. Register arrays are used to reshape the pooling input pixels and ensure continuous feeding of pixels into pooling PEs without idle cycles. The PEs are either comparators for max pooling or accumulators followed by constant coefficient multipliers for average pooling. The outputs of pooling are written back to the output pixel buffers and then transferred to the external memory.

D. Fully Connected Layers

The inner-product layer or fully connected (FC) layer is a special form of the convolution layer with $Nkx = Nky = Nox = Noy = 1$, or there are no Loop-1 and Loop-3. Therefore, we only unroll Loop-4 and reuse the same MAC unit array used in convolution layers for all the FC layers. Contrary to convolution layers, FC layers have large amount of weights but small amount of operations, which makes the throughput of FC layers primarily bounded by the off-chip communication speed. Due to this, dual weight buffers can be used to overlap the inner-product computation with off-chip communication to increase FC throughput. However, in recent CNN models, e.g., ResNet, the size of FC weights (= 2 M) has been significantly reduced compared to that of VGG (= 123.6M), and FC layers are completely removed in NiN [3]. Considering this trend that CNNs are decreasing their reliance on FC layers and the relatively smaller number of FC operations, the dual buffer techniques are not used in this paper. We reuse the convolution weight buffers for FC weights and start the FC computations after the weights are read from DRAM. Thus, in VGG implementation, FC layer has a significant contribution to the overall system latency. FC layer output pixels are directly stored in on-chip buffers as their size is small (<20 kB).

VII. Experimental Results

A. System Setup

The proposed hardware CNN inference accelerator is demonstrated by implementing NiN [3], VGG-16 [4], and ResNet-50/ResNet-152 [5] CNN models on two Intel FPGAs. The two Intel FPGAs, e.g., Stratix V GXA7 / Arria 10 GX 1150, consist of 234.7K/427.2K adaptive logic modules (ALMs), 256/1,518 DSP blocks and 2,560/2,713 M20K BRAM blocks, respectively. The underlying FPGA boards for Stratix V and Arria 10 are Terasic DE5-Net and Nallatech 385A, respectively, and both are equipped with two banks of 4GB DDR3 DRAMs.

The overall CNN acceleration system on the FPGA chip shown in Fig. 16 is coded in parametrized Verilog scripts and configured by the proposed CNN compiler in [16] for different CNN and FPGA pairs. If a layer does not exist in the CNN model, the corresponding computing module is not synthesized, and the dataflow just bypasses this module. With two DRAM banks, both kernel and feature maps are separated into these two banks to enable full off-chip communication. Two Modular scatter-gather DMA engines provided by Intel are used to simultaneously read and write from/to these two DRAM banks. Data scatter and gather [16] are used to distribute the data stream from DMA into multiple input buffers and collect data from multiple output buffers into one DMA stream, respectively. After the input images and weights are loaded into DRAMs, the CNN inference acceleration process starts. When the computation of one loop tile completes, the output pixels are transferred to DRAM, and then the weights and pixels for the next loop tile are loaded from DRAM to on-chip buffers. The controller governs the iterations of the four convolution loops and the layer-by-layer sequential computation. The buffer read and write addresses are also generated by the controller.

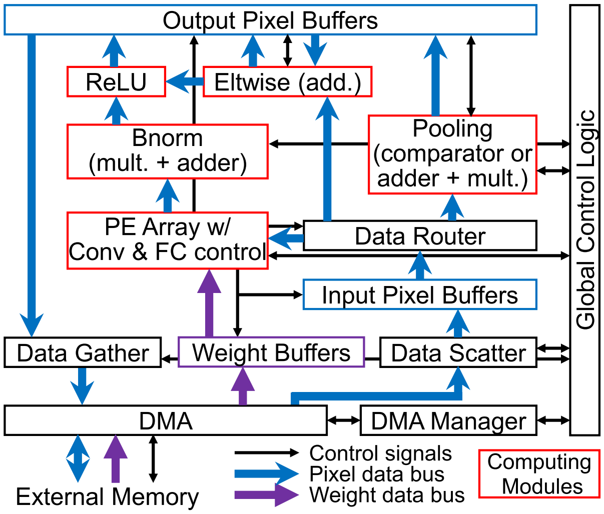


Fig. 16. Overall FPGA-based CNN hardware acceleration system [16].

The fixed-point data representation is used, and both pixels and weights are 16-bit. The decimal points are dynamically adjusted according to the ranges of pixel values in different layers to fully utilize the existing data width [13]. By this means, the top-1 and top-5 ImageNet classification accuracy degradation is within 2% compared with software floating-point implementation [10]–[14].

B. Analysis of Experimental Results

The performance and specifications of our proposed CNN accelerators are summarized in Table II. In Stratix V and Arria 10, one DSP block can be configured as either two independent 18-bit × 18-bit multipliers or one multiplier followed by an accumulator, i.e., one MAC. Since one multiplier consumes much more logic than one adder, we use the DSP as two independent multipliers and implement the accumulator inside the MAC unit by ALMs. Since Arria 10 has 1.8× more ALMs and 5.9× more DSP blocks than the Stratix V we use, larger loop unrolling variables (*Pox* × *Poy* × *Pof*) can be achieved in Arria 10 to obtain >2× throughput enhancement than Stratix V.

Authorized licensed use limited to: University of Louisiana at Lafayette. Downloaded on October 19,2020 at 23:35:00 UTC from IEEE Xplore. Restrictions apply.

TABLE III
PREVIOUS CNN FPGA IMPLEMENTATIONS

	[18]	[17]	[12]	[13]	[11]	[10]	[24]	[23]	[23]
CNN Model	AlexNet	AlexNet	AlexNet	VGG-16	VGG-16	VGG-16	VGG-16	VGG-19	ResNet-152
FPGA	Virtex-7 VC707	Virtex-7 VC709	Arria 10 GX 1150	Zynq XC7Z045	Stratix V GSD8	Virtex-7 VX690t	Arria 10 GT 1150	Stratix V GSMD5	Stratix V GSMD5
Clock (MHz)	160	156	303	150	120	150	231.85	150	150
# Operations (GOP)	1.33	1.46	1.46	30.76	30.95	30.95	30.95	39.26	22.62
# of Parameters	2.33 M	60.95 M	60.95 M	50.18 M	138.3 M	138.3 M	138.3 M	143.7 M	60.4 M
Precision	32bit fixed	16bit fixed	FP16bit	16bit fixed	8-16bit fixed	16bit fixed	8-16bit fixed	16bit fixed	16bit fixed
DSP Utilization	2,688 (96%)	2,144 (60%)	1,476 (97%)	780 (87%)	-	-	1,500 (99%)	1,044 (66%)	1,044 (66%)
Logic Element[a]	45K (9.2%)	274K (63%)	246K (58%)	183K (84%)	-	-	313K (73%)	45.7K (27%)	45.7K (27%)
On-chip Memory[b]	543 (53%)	956[b] (65%)	2,487 (92%)	486 (89%)	-	-	1668 (61%)	959 (48%)	959 (48%)
Latency/Image (ms)	-	8×2.56[c]	-	224.6	262.9	151.8	26.85	-	-
Throughput (GOPS)	147.82	565.94	1.38TFLOPS	136.97	117.8	203.9	1,171.3	364.36	226.47

[a] Xilinx FPGAs in LUTs and Intel FPGAs in ALMs
[b] Xilinx FPGAs in BRAMs (36 Kb) and Intel FPGAs in M20K RAMs (20 Kb)
[c] The reported delay of one pipeline stage is 2.56 ms and the number of pipeline stages equals 8.

Compared with [15], the unrolling variables, i.e., $Pox \times Poy \times Pof$, of VGG-16 are set to be $7 \times 7 \times 64$ on Arria 10 instead of $14 \times 14 \times 16$, where the number of MAC units ($= 3136$) are the same and both sets of $P*$ variables are the common factors of the feature/kernel map sizes resulting in the same computation cycles. The data router in Fig. 13 and the data buses after MAC units in Fig. 15 are only related with Pox and Poy, whereas the data buses related with Pof from weight buffers to MAC units in Fig. 15 are relatively simple. To reduce the data bus width and required logic, we choose smaller $Pox \times Poy$ in this work as 7×7 with a larger Pof as 64. Since the greatest common factors of feature/kernel maps, e.g., $Nox \times Noy \times Nof$, of all Convolution (Conv.) layers in ResNets are $7\times7\times64$, we still set $Pox \times Poy \times Pof$ to be $7\times7\times64$. Since ResNets have more complex structure and more types of layers, e.g., Eltwise and Bnorm, they consume more logic elements than NiN and VGG-16 on Arria 10 and cannot achieve the same parallel degree as NiN and VGG-16 on Stratix V. Since two FPGAs have close capacity of on-chip BRAMs, the loop tiling variables ($T*$) of the same CNN is set to be the same for both FPGAs, which leads to similar BRAM consumption.

The breakdown of the processing time per image of each CNN is shown in Fig. 17 with batch size = 1. The MAC computation time of convolution layers, e.g., "Conv MAC," dominates the total latency by over 50%. "Conv DRAM" includes DRAM transaction delay of convolution weights and input–output pixels. The FC latency includes the inner-product computation delay and the DRAM transfer delay of FC weights. "Others" include the delay of average pooling, element-wise and pipeline stages.

The logic utilization in ALMs of each module is shown in Fig. 18. Most multipliers in MAC units are implemented by DSPs, and logic elements are mainly used to implement accumulators in MAC units. With the same parallel computation degree, the MAC units of the four CNNs use about the same amount of ALMs. As VGG-16 is highly uniform with only one convolution sliding setting, e.g., stride = 1 and padding = 1, only one BUF2PE bus is needed, which leads to less logic and BRAM consumption of data router compared to NiN and ResNets. Convolution and FC layers share the MAC units but have their own control logic to govern the sequential operations. Eltwise layers use adders to element-wise add pixels from two branches of layers. "Others" include the system interconnections, global control logic, bias adders, and configuration registers.

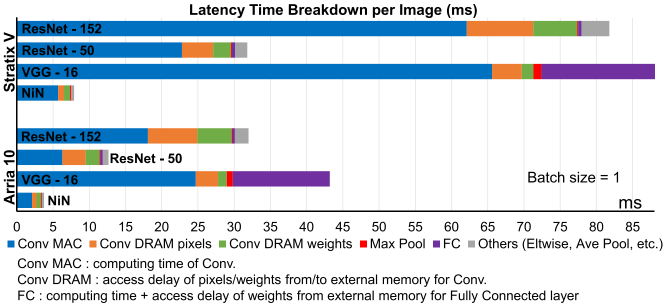


Fig. 17. Latency breakdown per image of NiN, VGG-16 and ResNet-50/152 on Stratix V and Arria 10 FPGA platforms [16].

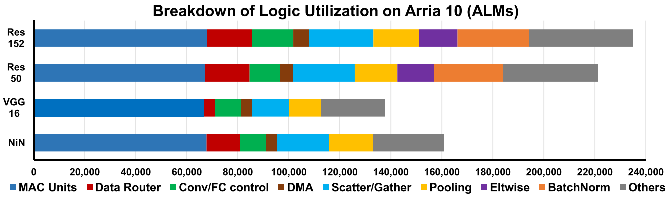


Fig. 18. Logic utilization breakdown of NiN, VGG-16 and ResNet-50/152.

C. Comparison With Prior Works

The reported results from recent CNN FPGA accelerators are listed in Table III. Rahman *et al.* [18] only implement convolution layers in AlexNet and uses the similar strategy as

Authorized licensed use limited to: University of Louisiana at Lafayette. Downloaded on October 19,2020 at 23:35:00 UTC from IEEE Xplore. Restrictions apply.

us to unroll Loop-3 and Loop-4, which can also achieve high DSP utilization. In [10] and [17], the layer-by-layer computation is pipelined using different part of one or multiple FPGAs resources to improve hardware utilization and thus throughput. However, with the highly increasing number of convolution layers [5], it becomes very difficult to map different layers onto different resources and balance the computation among all the pipeline stages. In addition, pipelining can increase the throughput but not necessarily the latency. Batch computing with multiple input images is applied in [8], [10], [12], [17], and [23]. The biggest advantage of this technique is to share the weights transferred from off-chip DRAM among multiple images and thus increase the throughput at the cost of increased latency per image and external memory storage of multiple images. Benefit from batch computing and using 2144 DSP slices, which enables high parallelism degree, Li *et al.* [17] also achieve high throughput of 565.94 GOPS for AlexNet. In [12], an OpenCL-based CNN accelerator is implemented on Arria10 FPGA, where the Intel FPGA SDK for OpenCL provides a pregenerated platform that ensures timing closure at higher frequency than our RTL design. The Winograd transform is applied for convolution layers that reduces multiplication operations by 2× or improves the throughput by 2× using the same number of DSPs. The 16-b floating-point data format is used with shared exponent, which allows directly using fixed-point 18-bit × 18-bit multipliers for floating-point operations. Wei *et al.* [24] proposed an OpenCL-based systolic array architecture to implement convolution on Arria 10, which reduces the global PE interconnect fan-out to achieve high frequency and resource utilization. The VGG-16 throughput of [24] is higher than ours mainly due to: 1) higher frequency; 2) lower precision of weights; and 3) dual buffer scheme to hide DRAM latency. Guan *et al.* [23] proposed an RTL–HLS hybrid framework to automatically generate FPGA hardware and implements convolution and FC as matrix multiplication. Although the Stratix-V GSMD5 (with 1590 DSP blocks) used in [23] has 6.2× more DSP blocks than our Stratix-V GXA7, our accelerator on Stratix V can realize 1.2× higher throughput for ResNet-152 by higher hardware (DSP and logic) utilization through the proposed loop optimization technique and exploiting logic elements to implement multipliers as well as DSPs.

VIII. Conclusion

In this paper, we present an in-depth analysis of convolution loop acceleration strategy by numerically characterizing the loop optimization techniques. The relationship between accelerator objectives and design variables are quantitatively investigated. A corresponding new dataflow and architecture is proposed to minimize data communication and enhance throughput. Our CNN accelerator implements end-to-end NiN, VGG-16, and ResNet-50/ResNet-152 CNN models on Stratix V and Arria 10 FPGA, achieving the overall throughput of 348 GOPS and 715 GOPS, respectively.

References

[1] O. Russakovsky *et al.*, “ImageNet large scale visual recognition challenge,” *Int. J. Comput. Vis.*, vol. 115, no. 3, pp. 211–252, 2015.

[2] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “ImageNet classification with deep convolutional neural networks,” in *Proc. Adv. Neural Inf. Process. Syst. (NIPS)*, 2012, pp. 1097–1105.

[3] M. Lin, Q. Chen, and S. Yan. (Mar. 2014). “Network in network.” [Online]. Available: https://arxiv.org/abs/1312.4400

[4] K. Simonyan and A. Zisserman, “Very deep convolutional networks for large-scale image recognition,” in *Proc. Int. Conf. Learn. Represent. (ICLR)*, 2015.

[5] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2016, pp. 770–778.

[6] D. F. Bacon, S. L. Graham, and O. J. Sharp, “Compiler transformations for high-performance computing,” *ACM Comput. Surv.*, vol. 26, no. 4, pp. 345–420, Dec. 1994.

[7] 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. 51, no. 1, pp. 127–138, Jan. 2017.

[8] Y.-H. Chen, J. Emer, and V. Sze, “Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks,” in *Proc. ACM/IEEE Int. Symp. Comput. Archit. (ISCA)*, Jun. 2016, pp. 367–379.

[9] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, “Optimizing FPGA-based accelerator design for deep convolutional neural networks,” in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays (FPGA)*, Feb. 2015, pp. 161–170.

[10] C. Zhang, D. Wu, J. Sun, G. Sun, G. Luo, and J. Cong, “Energy-efficient CNN implementation on a deeply pipelined FPGA cluster,” in *Proc. ACM Int. Symp. Low Power Electron. Design (ISLPED)*, Aug. 2016, pp. 326–331.

[11] N. Suda *et al.*, “Throughput-optimized OpenCL-based FPGA accelerator for large-scale convolutional neural networks,” in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays (FPGA)*, Feb. 2016, pp. 16–25.

[12] U. Aydonat, S. O’Connell, D. Capalija, A. C. Ling, and G. R. Chiu, “An OpenCL deep learning accelerator on Arria 10,” in *Proc. ACM/SIGDA Symp. Field-Program. Gate Arrays (FPGA)*, Feb. 2017, pp. 55–64.

[13] K. Guo *et al.*, “Angel-Eye: A complete design flow for mapping CNN onto embedded FPGA,” *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 37, no. 1, pp. 35–47, Jan. 2018.

[14] Y. Ma, N. Suda, Y. Cao, J.-S. Seo, and S. Vrudhula, “Scalable and modularized RTL compilation of convolutional neural networks onto FPGA,” in *Proc. IEEE Int. Conf. Field-Program. Logic Appl. (FPL)*, Aug./Sep. 2016, pp. 1–8.

[15] Y. Ma, Y. Cao, S. Vrudhula, and J.-S. Seo, “Optimizing loop operation and dataflow in FPGA acceleration of deep convolutional neural networks,” in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays (FPGA)*, Feb. 2017, pp. 45–54.

[16] Y. Ma, Y. Cao, S. Vrudhula, and J.-S. Seo, “An automatic RTL compiler for high-throughput FPGA implementation of diverse deep convolutional neural networks,” in *Proc. IEEE Int. Conf. Field-Program. Logic Appl. (FPL)*, Sep. 2017, pp. 1–8.

[17] 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 *Proc. IEEE Int. Conf. Field-Program. Logic Appl. (FPL)*, Aug. 2016, pp. 1–9.

[18] A. Rahman, J. Lee, and K. Choi, “Efficient FPGA acceleration of convolutional neural networks using logical-3D compute array,” in *Proc. IEEE Design, Auto. Test Eur. Conf. (DATE)*, Mar. 2016, pp. 1393–1398.

[19] M. Motamedi, P. Gysel, V. Akella, and S. Ghiasi, “Design space exploration of FPGA-based deep convolutional neural networks,” in *Proc. IEEE Asia South Pacific Design Auto. Conf. (ASP-DAC)*, Jan. 2016, pp. 575–580.

[20] S. Han *et al.*, “EIE: Efficient inference engine on compressed deep neural network,” in *Proc. ACM/IEEE Int. Symp. Comput. Archit. (ISCA)*, Jun. 2016, pp. 243–254.

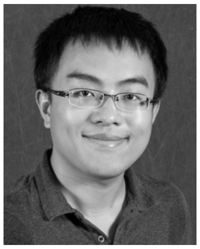
[21] L. Du *et al.*, “A reconfigurable streaming deep convolutional neural network accelerator for Internet of Things,” *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 65, no. 1, pp. 198–208, Jan. 2018.

[22] B. Bosi, G. Bois, and Y. Savaria, “Reconfigurable pipelined 2-D convolvers for fast digital signal processing,” *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 7, no. 3, pp. 299–308, Sep. 1999.

[23] Y. Guan *et al.*, “FP-DNN: an automated framework for mapping deep neural networks onto FPGAs with RTL-HLS hybrid templates,” in *Proc. IEEE Int. Symp. Field-Program. Custom Comput. Mach. (FCCM)*, Apr./May 2017, pp. 152–159.

[24] X. Wei *et al.*, “Automated systolic array architecture synthesis for high throughput CNN inference on FPGAs,” in *Proc. ACM the 54th Annu. Design Autom. Conf. (DAC)*, Jun. 2017, pp. 1–6.

Authorized licensed use limited to: University of Louisiana at Lafayette. Downloaded on October 19,2020 at 23:35:00 UTC from IEEE Xplore. Restrictions apply.



Yufei Ma (S'16) received the B.S. degree in information engineering from the Nanjing University of Aeronautics and Astronautics, Nanjing, China, in 2011 and the M.S.E. degree in electrical engineering from the University of Pennsylvania, Philadelphia, PA, USA, in 2013. He is currently working toward the Ph.D. degree at Arizona State University, Tempe, AZ, USA.

His current research interests include the high-performance hardware acceleration of deep learning algorithms on digital application-specified integrated circuit and field-programmable gate arrays.

Yu Cao (S'99–M'02–SM'09–F'17) received the B.S. degree in physics from Peking University, Beijing, China, in 1996 and the M.A. degree in biophysics and the Ph.D. degree in electrical engineering from the University of California, Berkeley, CA, USA, in 1999 and 2002, respectively.

He was a Summer Intern at Hewlett-Packard Labs, Palo Alto, CA, USA, in 2000, and at the IBM Microelectronics Division, East Fishkill, NY, USA, in 2001. He was a Postdoctoral Researcher at the Berkeley Wireless Research Center, University of California. He is currently a Professor of Electrical Engineering at Arizona State University, Tempe, AZ, USA. He has authored or coauthored numerous articles and two books on *Nano-CMOS Modeling and Physical Design*. His current research interests include physical modeling of nanoscale technologies, design solutions for variability and reliability, reliable integration of postsilicon technologies, and hardware designs for on-chip learning.

Dr. Cao was a recipient of the 2012 Best Paper Award at the IEEE Computer Society Annual Symposium on VLSI, the 2010, 2012, 2013, 2015, and 2016 Top 5% Teaching Award, Schools of Engineering, Arizona State University, the 2009 ACM SIGDA Outstanding New Faculty Award, the 2009 Promotion and Tenure Faculty Exemplar, Arizona State University, the 2009 Distinguished Lecturer of the IEEE Circuits and Systems Society, the 2008 Chunhui Award for outstanding oversea Chinese scholars, the 2007 Best Paper Award at International Symposium on Low Power Electronics and Design, the 2006 NSF CAREER Award, the 2006 and 2007 IBM Faculty Award, the 2004 Best Paper Award at International Symposium on Quality Electronic Design, and the 2000 Beatrice Winner Award at International Solid-State Circuits Conference. He was an Associate Editor of the IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems. He served on the technical program committee of many conferences.

Sarma Vrudhula (M'85–SM'02–F'16) received the B.Math. degree from the University of Waterloo, Waterloo, ON, Canada, and the M.S.E.E. and Ph.D. degrees in electrical and computer engineering from the University of Southern California, Los Angeles, CA, USA.

He was a Professor at the ECE Department, University of Arizona, Tucson AZ, USA, and was on the faculty of the EE-Systems Department at the University of Southern California. He was also the Founding Director of the NSF Center for Low Power Electronics at the University of Arizona. He is currently a Professor of Computer Science and Engineering with Arizona State University, Tempe, AZ, USA, and the Director of the NSF I/UCRC Center for Embedded Systems. His current research interests include design automation and computer aided design for digital integrated circuit and systems; low-power circuit design; energy management of circuits and systems; energy optimization of battery powered computing systems, including smartphones, wireless sensor networks, and Internet of Things systems that relies energy harvesting; system level dynamic power and thermal management of multicore processors and system-on-chip; statistical methods for the analysis of process variations; statistical optimization of performance, power, and leakage; a new circuit architectures of threshold logic circuits for the design of application-specific integrated circuits and field-programmable gate arrays; nonconventional methods for implementing logic, including technology mapping with threshold logic circuits; the implementation of threshold logic using resistive memory devices; and the design and optimization of nonvolatile logic.

Jae-sun Seo (S'04–M'10–SM'17) received the B.S. degree in electrical engineering from Seoul National University, Seoul, South Korea, in 2001 and the M.S. and Ph.D. degrees in electrical engineering from the University of Michigan, Ann Arbor, MI, USA, in 2006 and 2010, respectively.

From 2010 to 2013, he was with the IBM T. J. Watson Research Center, Yorktown Heights, NY, USA, focused on the cognitive computing chips under the DARPA SyNAPSE Project and energy-efficient integrated circuits for high-performance processors. In 2014, he joined the School of Electrical, Computer and Energy Engineering, Arizona State University, Tempe, AZ, USA, as an Assistant Professor. In 2015, he was with the Intel Circuits Research Labortory as a Visiting Faculty. His current research interests include efficient hardware design of machine learning and neuromorphic algorithms and integrated power management.

Dr. Seo was a recipient of the Samsung Scholarship from 2004 to 2009, the IBM Outstanding Technical Achievement Award in 2012, and the NSF CAREER Award in 2017.

Authorized licensed use limited to: University of Louisiana at Lafayette. Downloaded on October 19,2020 at 23:35:00 UTC from IEEE Xplore. Restrictions apply.