Optimizing Loop Operation and Dataflow in FPGA Acceleration of Deep Convolutional Neural Networks

Yufei Ma, Yu Cao, Sarma Vrudhula†, Jae-sun Seo
School of Electrical, Computer and Energy Engineering
†School of Computing, Informatics, Decision Systems Engineering
Arizona State University, Tempe, USA
{yufeima, yu.cao, vrudhula, jaesun.seo}@asu.edu

ABSTRACT
As convolution layers contribute most operations in convolutional neural network (CNN) algorithms, an effective convolution acceleration scheme significantly affects the efficiency and performance of a hardware CNN accelerator. Convolution in CNNs involves three-dimensional multiply and accumulate (MAC) 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 work overcomes these barriers by quantitatively analyzing and optimizing the design objectives (e.g. required memory access) of the CNN accelerator based on multiple design variables. We systematically explore the trade-offs of hardware cost by searching the design variable configurations, and propose a specific dataflow of hardware CNN acceleration to minimize the memory access and data movement while maximizing the resource utilization to achieve high performance. The proposed CNN acceleration scheme and architecture are demonstrated on a standalone Altera Arria 10 GX 1150 FPGA by implementing end-to-end VGG-16 CNN model and achieved 645.25 GOPS of throughput and 47.97 ms of latency, which is a >3.2× enhancement compared to state-of-the-art FPGA implementations of VGG model.

Keywords
Convolutional neural networks; FPGA; hardware acceleration.

1. INTRODUCTION
FPGA-based CNN accelerators are gaining popularity because of higher energy efficiency than GPUs [8-12], greater flexibility due to reconfigurability and shorter turn-around time than ASICs. They also allow low latency operation by enabling the customization of the acceleration architecture and utilizing of hundreds to thousands of on-chip DSP blocks. The large number of operations (>1G) and kernel weights (>50M) in the state-of-the-art deep CNN algorithms have become a well-known challenge for their implementations on embedded platforms, which are constrained by limited hardware computing resources and costly off-chip communication. In addition, the high variability in the sizes of the different convolution layers may prevent the full utilization of the available computing resources for all the layers. Moreover, the increasing scale and complexity of CNN algorithms to achieve higher accuracy further exacerbate the energy consumption for computation, communication and memory. In fact, the energy cost of the increased data movement and memory accesses due to the large number of operations often exceeds the energy cost of computation [6][15]. Therefore, an energy-efficient accelerator should target fully utilizing computation resources as well as minimizing data movement and memory access.

Since convolution operations often constitute more than 90% of the total CNN operations [2][3][4][6][11], an effective acceleration scheme of convolution requires proper management of the parallel computations and the organization of data storage and access across multiple levels of memories, e.g. off-chip DRAM, on-chip SRAM and local registers. Convolution is comprised of four levels of loops sliding along both kernel and feature maps, resulting in a large design space to explore 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 [7][17], such as loop unrolling, tiling and interchange. Although some CNN accelerators have adopted these techniques [7][9][10][14], the impact of these techniques on design efficiency and performance has not been systematically studied. Instead, most prior works only explore the design space after the hardware architecture or the parallelism scheme has been fixed, and optimize their implementation by only tuning the design variables within their architecture. Without fully studying the loop operations of three-dimensional convolutions, it is difficult to efficiently customize the dataflow and architecture for high-throughput CNN implementations.

In this paper, we provide an in-depth analysis of the three loop optimization techniques for convolution operations and also use corresponding design variables to numerically characterize the acceleration scheme. By this means, the design objectives of CNN accelerators can be quantitatively estimated based on the configurations of the design variables. By searching through the configurations of design variables that minimize the estimated objectives, an efficient dataflow aimed at minimizing the data movements and memory access is determined. Then a corresponding architecture is also designed that fully utilizes the computing resources for high performance, which is uniform and reusable for all the layers. The proposed acceleration scheme and architecture was validated by implementing a large-scale CNN algorithm, VGG-16 [3] for image recognition [1], on the Altera Arria 10 GX 1150 FPGA, and demonstrated throughput of 645.25 GOPS and end-to-end latency of 47.97 ms.

FPGA'17, February 22–24, 2017, Monterey, CA, USA.

DOI: http://dx.doi.org/10.1145/3020078.3021736

```
for (no = 0; no < Nof; no ++)  ──────────►  Loop-4
    for (y = 0; y < Noy; y =+ S)
        for (x = 0; x < Nox; x =+ S)   ──────────►  Loop-3
            for (ni = 0; ni < Nif; ni ++)  ──────────►  Loop-2
                for (ky = 0; ky < Nky; ky ++)
                    for (kx = 0; kx < Nkx; kx ++)   ──────────►  Loop-1
                        pixel_L(no; x, y) += pixel_L-1(ni; x + kx, y + ky) × weight_L-1(ni, no; kx, ky);
    pixel_L (no; x, y) = pixel_L(no; x, y) + bias(no);
```

Figure 1. Four levels of convolution loops, where *L* denotes the index of convolution layer and S denotes the sliding stride.

The rest of the paper is organized as follows. Section 2 proposes the key design variables that are used to numerically characterize the loop optimization techniques for accelerating the convolution loops. Section 3 describes the quantitatively analysis and estimation of hardware accelerator objectives based on the loop optimizing design variables. Section 4 discusses the acceleration schemes used in recent state-of-the-art CNN accelerators. Section 5 presents the optimization process of minimizing the design objectives and the optimized acceleration scheme with specific design variables. A corresponding dataflow and computing architecture is proposed in Section 6 for accelerating convolution and pooling operations. Section 7 analyzes the experimental results and compares with prior works. Conclusions are presented in Section 8.

2. ACCELERATION OF CONVOLUTION LOOPS

2.1 General CNN Accelerator System

As recent deep CNN algorithms involve a large amount of data and weights, on-chip memory is usually insufficient to store the entire data, requiring external memory with ~GB capacity. Therefore, a typical CNN accelerator consists of three levels of hierarchy: 1) external memory, 2) on-chip buffers, and 3) register files and processing engines (PEs). The basic flow is to fetch data and weights 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 the next layer inputs.

2.2 Convolution Loops

Convolution is the main operation in CNN algorithms, which involves three-dimensional multiply and accumulate (MAC) operations of input feature maps and convolution kernel weights. Convolution is comprised of four levels of loops as shown in the pseudo codes in Figure 1 and illustrated in Figure 2. To efficiently map and perform the convolution loops, three loop optimization techniques [7][17], 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.

Loop unrolling determines the parallelism scheme of certain convolution loops, and thus the required size of registers and PEs. Loop tiling determines the required capacity of on-chip buffers. It divides the loops into multiple blocks, and the data of the executing block are read from external memory and stored in on-chip buffers. Loop interchange determines the computation order of the four loops and thus affects the dataflow between the adjacent levels of memory hierarchy.

2.3 Loop Optimization and Design Variables

As shown in Figure 2, multiple parameters are used to describe the dimensions of the feature and kernel maps of each convolution layer for a given CNN model. For the given set of parameters, the hardware design variables of loop unrolling and loop tiling will determine the acceleration factor and hardware footprint. All parameters and variables used in this work are listed in Table 1.

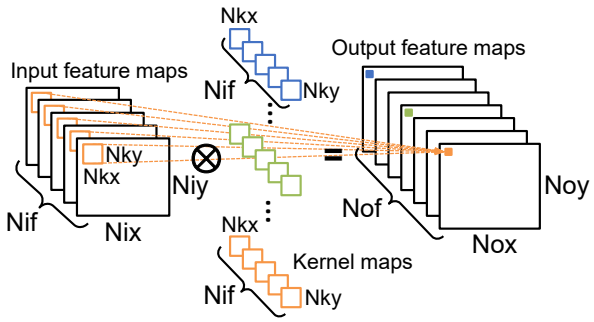

Figure 2. Convolution operation and its parameters.

The width and height of one kernel window is described by (Nkx, Nky). (Nix, Niy) and (Nox, Noy) define the width and height of one input and output feature map, 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 along different feature or kernel map dimensions. 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 local or on-chip buffers. The constraints of these parameters and variables are given by $1 \le P^* \le T^* \le N^*$, where N*, *T** and *P** denote any parameter or variable that has a prefix of capital N, *T* and *P*, respectively. For

Table 1. Convolution Loop Parameters and 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	Nkx	Nky	Nix	Niy	Nox	Noy	Nif	Nof
Loop Tiling	*Tkx*	*Tky*	*Tix*	*Tiy*	*Tox*	*Toy*	*Tif*	*Tof*
Loop Unrolling	*Pkx*	*Pky*	*Pix*	*Piy*	*Pox*	*Poy*	*Pif*	*Pof*

instance, $1 \le Pkx \le Tkx \le$ Nkx. By default, P^*, T^* and N* are applied to all convolution layers.

The relationship of input and output variables can be computed by Equations (1)-(3), where S is the stride of the sliding window and the zero padding size is included in Nix, Niy, *Tix* and *Tiy*.

$$\begin{aligned} \text{Nix} &= (\text{Nox} - 1)\text{S} + \text{Nkx} \\ \text{Niy} &= (\text{Noy} - 1)\text{S} + \text{Nky} \end{aligned} \quad (1)$$

$$\begin{aligned} Tix &= (Tox - 1)\text{S} + \text{Nkx} \\ Tiy &= (Toy - 1)\text{S} + \text{Nky} \end{aligned} \quad (2)$$

$$\begin{aligned} Pix &= Pox \\ Piy &= Poy \end{aligned} \quad (3)$$

The parameters or variables (N*, T^*, P^*) determine the configurations of the three levels of memory hierarchy from external memory to on-chip buffers to registers and PEs.

2.3.1 Loop Unrolling

As illustrated in Figure 3, unrolling different convolution loops directs parallelization of different computations, which affects the optimal PE array architecture with respect to the data reuse opportunities and memory access patterns.

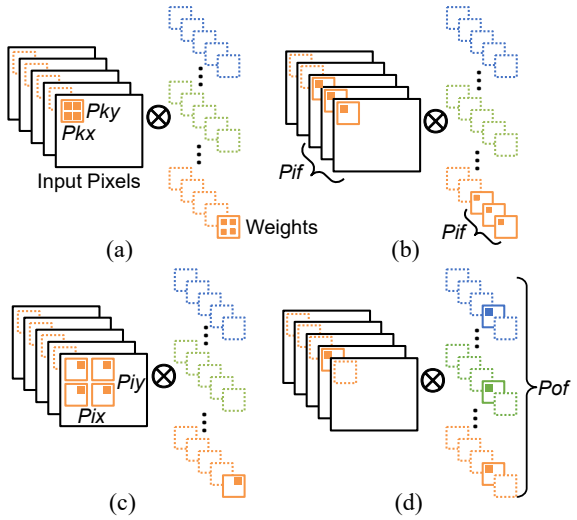

Figure 3. Loop unrolling: (a) unroll Loop-1; (b) unroll Loop-2; (c) unroll Loop-3; (d) unroll Loop-4.

Loop-1 unrolled (Figure 3(a)): in this case, the inner product of $Pkx \times Pky$ pixels 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 $Pkx \times Pky$ to sum the $Pkx \times Pky$ parallel multiplication results, and an accumulator to add the adder tree output with the previous partial sum.

Loop-2 unrolled (Figure 3(b)): 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*.

Loop-3 unrolled (Figure 3(c)): in every cycle, $Pix \times Piy$ number of pixels from different (x, y) locations in the same feature map are multiplied with the identical weight, or this weight can be reused by $Pix \times Piy$ times. Since the $Pix \times Piy$ parallel multiplication contributes to independent $Pix \times Piy$ output pixels, $Pix \times Piy$ accumulators are used to serially accumulate the multiplier outputs and no adder tree is needed.

Loop-4 unrolled (Figure 3(d)): in every cycle, one pixel is multiplied with *Pof* weights at the same (x, y) location but from *Pof* different kernel maps, and this pixel is reused by *Pof* times. The computing structure is identical to the case of unrolling Loop-3 using *Pof* multipliers and accumulators and no adder tree.

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

$$Pm = Pkx \times Pky \times Pif \times Pix \times Piy \times Pof. \quad (4)$$

2.3.2 Loop Tiling

On-chip memory of FPGAs is not always large enough to store the entire data of deep CNN algorithms altogether. 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 fit into the on-chip buffers. With proper assignments of the loop tiling size, the locality of data can be increased to reduce the number of external memory 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 $Tix \times Tiy \times Tif \times$ (pixel_datawidth). The size of weight buffer is $Tkx \times Tky \times Tif \times Tof \times$ (weight_datawidth). The size of output pixel buffer is $Tox \times Toy \times Tof \times$ (pixel_datawidth).

2.3.3 Loop Interchange

Loop interchange determines the sequential computation order of the four convolution loops. There are two kinds of loop interchange, namely intra-tiling and inter-tiling loop orders. Intra-tiling loop order determines the pattern of data movements from on-chip buffer to register files or PEs. Inter-tiling loop order determines the data movement from external memory to on-chip buffer. The innermost loop is computed first and the outermost loop is computed at the end. Loops that are fully unrolled within the tiling block ($P^* = T^*$) are set to be the innermost intra-tiling loops by default. Loops with tiling size covering the full loop dimension ($T^* =$ N*) are the innermost inter-tiling loops by default.

3. ANALYSIS ON DESIGN OBJECTIVES OF CNN ACCELERATOR

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

1) *Computing latency* depends strongly on the loop unrolling factors P^*, but can also be affected by inefficient utilization of PEs and external memory transactions.
2) The requirement of *partial sum storage* is mainly determined by the order of loop computation. The earlier the final pixel output can be obtained, the fewer the number of partial sums that needs to be stored.
3) To reduce *the number of on-chip buffer accesses*, the pixels and weights fetched from the on-chip buffer need to be reused as much as possible, which is largely determined by the loop unrolling strategy.
4) *The number of external memory accesses* primarily relies on the size of on-chip buffers, which is determined by the loop tiling variables T^*.

3.1 Computing Latency

The number of multiplication operations per layer (Nm) is

$$\text{Nm} = \text{Nif} \times \text{Nkx} \times \text{Nky} \times \text{Nof} \times \text{Nox} \times \text{Noy}. \quad (5)$$

Ideally, the number of computing cycles per layer should be Nm/*Pm*. 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 (*#_cycles*) is

$$\#\_cycles = \#inter\text{-}tiling\_cycles \times \#intra\text{-}tiling\_cycles, \quad (6)$$

where

$$\#inter\text{-}tiling\_cycles = \lceil Nif/Tif \rceil \lceil Nkx/Tkx \rceil \lceil Nky/Tky \rceil \lceil Nof/Tof \rceil \lceil Nox/Tox \rceil \lceil Noy/Toy \rceil, \quad (7)$$

$$\#intra\text{-}tiling\_cycles = \lceil Tif/Pif \rceil \lceil Tkx/Pkx \rceil \lceil Tky/Pky \rceil \lceil Tof/Pof \rceil \lceil Tox/Pox \rceil \lceil Toy/Poy \rceil. \quad (8)$$

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.

3.2 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 flow chart to calculate the number of partial sums stored in memory (*#psum*) is shown in Figure 4. 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, there are no partial sums that need to be stored. If the loop tile size can cover all pixels and weights used in Loop-1 (*Tkx* = Nkx & *Tky* = Nky) and Loop-2 (*Tif* = Nif), then the partial sums can be consumed within this tile as described in Equations (4.2) – (4.5) inside Figure 4. In this case, the number of partial sums, determined by *P** or *T**, is small and can be stored in local registers (Equation (4.2)) or at most in on-chip buffers (Equation (4.3)). 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 Equations (4.6) – (4.9) inside Figure 4. In this case, the partial sums need to be stored at least in on-chip buffers (Equation (4.6)) or even in external memory (Equation (4.7)). The loop computing order also affects the number of partial sums, and the earlier Loop-1 and Loop-2 are computed, the fewer are 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 [6], since partial sums involve both read and write memory operations and typically require higher precision than pixels and weights.

3.3 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 clock 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:

$$Pwt = Pof \times Pif \times Pkx \times Pky \quad (9)$$

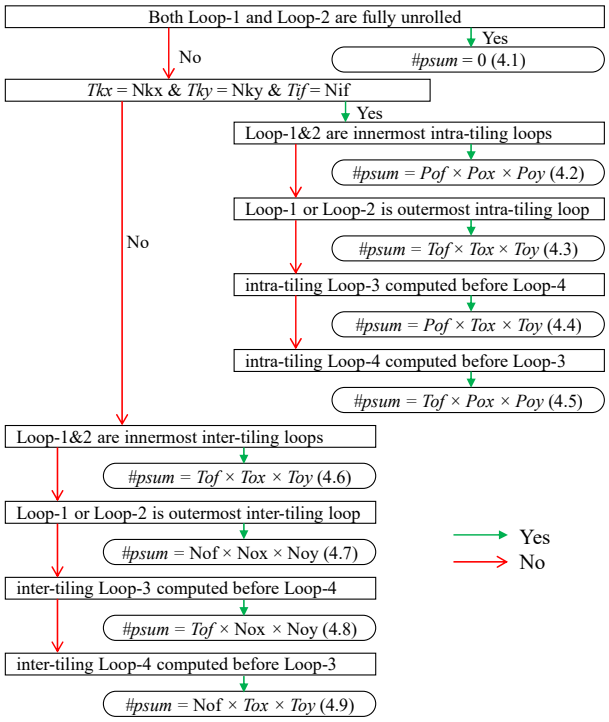

Figure 4. Flow chart that determines the total number of partial sums that needs to be stored in memory.

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

$$Ppx = Pif \times Pix \times Piy \quad (10)$$

Otherwise, *Ppx* is:

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

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:

$$Reuse\_wt = Pm / Pwt = Pix \times Piy \quad (12)$$

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:

$$Reuse\_px = Pm / Ppx \quad (13)$$

If Loop-1 is not unrolled, *Reuse_px* is:

$$Reuse\_px = Pof \quad (14)$$

otherwise, *Reuse_px* is:

$$Reuse\_px = \frac{Pof \times Pkx \times Pky \times Pix \times Piy}{((Pix-1)S + Pkx) \times ((Piy-1)S + Pky)} \quad (15)$$

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 Loop-3 is the innermost intra-tiling loop, the weights can be reused for *Tox*×*Toy*/(*Pox*×*Poy*) consecutive cycles. If Loop-4 is the innermost intra-tiling loop, the pixels can be reused for *Tof*/*Pof* consecutive cycles.

3.4 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:

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

and the total number of read operations of buffers for pixels is:

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

If the final output pixels cannot be obtained within one tile, their partial sums are stored in on-chip buffers. The number of write and read operations to/from on-chip buffers for partial sums per cycle is $2 \times Pof \times Pox \times Poy$, where all partial sums generated by Loop-1 (*Pkx*, *Pky*) and Loop-2 (*Pif*) are already summed together right after multiplications. The total number of write and read operations to/from on-chip 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:

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

3.5 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 block memory accesses [6][15][16], 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 flow chart to estimate the number of DRAM accesses is shown in Figure 5, 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 Equation (5.8) inside Figure 5. By computing Loop-3 first, weights stored in buffer are reused and *#DRAM_wt* is reduced as in Equation (5.1) and (5.5). Similarly, by computing Loop-4 first, pixels can be reused to reduce *#DRAM_px* as in Equation (5.3) and (5.6). 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.

The DRAM access of output pixels is not considered in the analysis because it is constant as every output pixel is written to DRAM only once. As Nkx > S, there are overlaps of pixels on the boundary of two tiles, and these pixels may be read twice by the two tiles. Since the number of the additional read is negligible, we do not include them in the analysis.

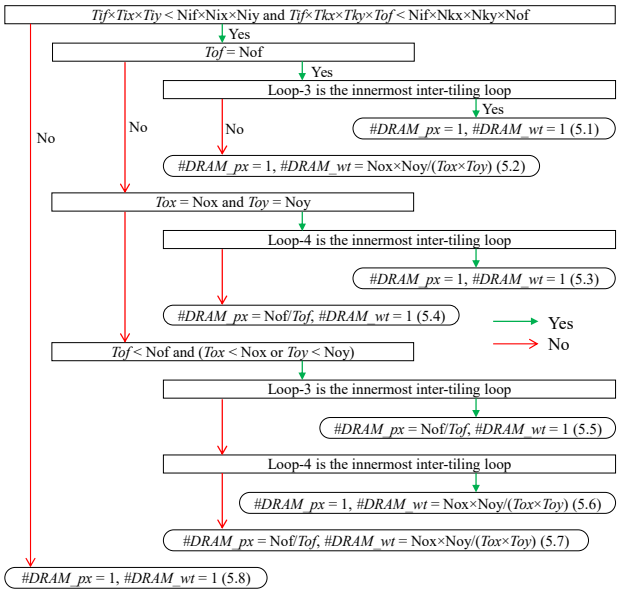

Figure 5. Flow chart that determines the average number of DRAM accesses of a single pixel and a single weight.

4. 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:

(A) Unroll Loop-1, Loop-2, Loop-4 [9][10][12][14]
(B) Unroll Loop-2, Loop-4 [7][11]
(C) Unroll Loop-1, Loop-3 [5][6]
(D) Unroll Loop-3, Loop-4 [13]

By unrolling Loop-1, Loop-2 and Loop-4 in type-(A), parallelism is employed in kernel maps, input feature maps and output feature maps. However, kernel window 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 the size of kernel window may vary considerably across different convolution layers in a given CNN model (e.g., AlexNet [2], ResidualNet [4]), which may cause workload imbalance and inefficient utilization of the PEs. To address this, PEs need to be configured differently for layers with different kernel window sizes [8], which increases the control complexity.

In type-(C), every row in the kernel window is fully unrolled (*Pkx* = Nkx) and Loop-3 is also partially unrolled [5][6]. By this means, pixels can be reused by the overlapping caused by Loop-1 and Loop-3 as in Equation (15), and weight reuse can also be realized by unrolling Loop-3 as in Equation (12). However, Loop-4 is not unrolled and further pixel reuse cannot be achieved. The issue caused by unrolling Loop-1 also affects type-(C).

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.

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 (≥ 13×13×64) is very large across all the convolution layers in AlexNet [2] and VGG [3] so that high level of parallelism can be

achieved even for largest FPGA available with ~3,600 DSP blocks. By this means, a uniform configuration and structure of PEs can be applied for all the convolution layers. [13] unrolled Loop-3 and Loop-4, however the choice and impact of such parallelism scheme is not quantitatively analyzed.

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 [10][13] have shown their tiling configurations that determine the on-chip buffer size, and the trade-off between the loop tiling size and the number of external memory accesses is not explored. The tiling size in [13] does not cover Loop-1 and Loop-2, e.g., $Tkx = Tky = Tif = 1$, which could significantly increase the number and movements of partial sums.

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.

5. PROPOSED ACCELERATION SCHEME

Based on the design objectives and analysis in Section 3, the optimization process of our acceleration scheme is presented in this section, which includes appropriate selection of the convolution loop design variables.

5.1 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 $\lceil N^*/T^* \rceil - N^*/T^*$ and $\lceil T^*/P^* \rceil - T^*/P^*$ as small as possible to minimize the inefficiency caused by the difference in sizes of CNN models. Based on the number of DSP blocks in our Arria 10 FPGA (= 1,518), we target the number of parallel multiplications Pm to be around 3,000 (one DSP block supports two 18-bit×18-bit multiplications), to fully utilize the available computing resources and minimize the latency.

5.2 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 4 and maximize the data reuse as discussed in Section 3.3, we decide to unroll Loop-3 (Pox>1 or Poy>1) and Loop-4 (Pof >1). By this means, we cannot attain zero partial sum storage as Equation (4.1) inside Figure 4.

Constrained by $1 \le P^* \le T^* \le N^*$, the least number of partial sum storage is achieved by Equation (4.2) among Equations (4.2) – (4.9) inside Figure 4. To satisfy the condition for Equation (4.2), 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 can achieve $Pof \times Pox \times Poy$ number of partial sums, which can be retained in local registers with minimum data movements.

5.3 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 Equation (12) and unrolling Loop-4 to reuse pixels as shown in Equation (14). By keeping the data used in Loop-1 in local registers, the sliding window across overlapped pixels can also be reused, similar to Equation (15), which unrolls both Loop-1 and Loop-3. As our partial sums are kept on local registers, they do not add overhead to the buffer access and storage.

5.4 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 Equation (5.1) and (5.3) inside Figure 5, 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 Equation (5.8) inside Figure 5.

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

$$\begin{aligned} &\textbf{minimize } bits\_BUF\_px\_wt \\ &\textbf{subject to } \#Tile\_px_L = 1 \text{ or } \#Tile\_wt_L = 1 \\ &\textbf{with } \forall\, L \in [1, \#CONVs], \end{aligned} \tag{20}$$

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

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

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

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 Equation (20) is also determined by T^* variables,

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

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

By solving Equation (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 5.2, we can only achieve a sub-optimal 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 stored in the on-chip buffer to benefit the DMA transactions with continuous data.

Finally, we have to solve Equation (20) by searching Toy and Tof, because it has a non-linear objective function and constraints with integer variables. Since Toy and Tof in VGG-16 consist of 2 × #CONVs = 26 variables and some variables can have up to 6 candidate values constrained by T^*/P^* = power of 2 and $T^* \le N^*$, the total number of Toy and Tof configurations is 7.2×10^{13}, which becomes an enormous solution space. Therefore, we only randomly sample ~3.6×10^9 configurations, which takes the MATLAB scripts to run for ~5 hours on two desktops. The obtained minimum solution for $bits\_BUF\_px\_wt$ results in an on-chip buffer size of 20.8 Mbits with pixel_datawidth = 16 and weight_datawidth = 8. As 20.8 Mbits is already <50% of our FPGA on-chip RAM

capacity, we did not continue the optimizing process that may lead to a better solution than 20.8 Mbits.

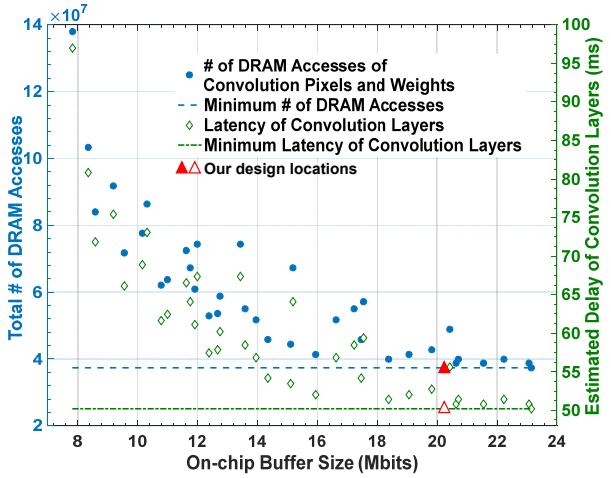

Figure 6. As a function of on-chip buffer size, the total number of DRAM accesses and estimated delay of convolution layers are shown.

If the configurations of *Tkx* = Nkx, *Tky* = Nky, *Tif* = Nif and *Tox* = Noy are kept, a smaller on-chip buffer can be used but the number of DRAM accesses will increase, as illustrated in Figure 6. The number of DRAM accesses in Figure 6 are obtained by randomly sampling ~5.1×10^9 *Toy* and *Tof* configurations using ~9 hours on the same desktops, and only the minimum result is kept for a specific interval of buffer size. The estimated theoretical delay of convolution layers in Figure 6 includes computing latency with 3,136 parallel MAC operations at 150MHz and DRAM transfer delay, which we assumed as 0.46ns per DRAM access. The minimum on-chip buffer requirement is 7.8 Mbits by minimizing *Toy* and *Tof*. If 7.8 Mbits still cannot meet the on-chip memory capacity, Equation (20) should be solved without the *Tox* = Nox constraint, or change the loop tiling strategy.

5.5 Optimized Loop Design Variables

According to the aforementioned optimization process, we propose an acceleration scheme for a high-performance and low-communication CNN accelerator.

5.5.1 Loop Unrolling

For all the convolution layers, we set uniform unrolling factors as *Pkx* = 1, *Pky* = 1, *Pif* = 1, *Pox* = 14, *Poy* =14 and *Pof* = 16, which enables *Pm* = 3,136 parallel multiplications, based on Section 5.1 5.2 and 5.3. 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.

5.5.2 Loop Tiling

For loop tiling, we set *Tkx* = Nkx, *Tky* = Nky, *Tif* = Nif as discussed in Section 5.2 so that data used in Loop-1 and Loop-2 are all buffered and *Tox* = Nox to benefit DMA transaction. Details of *Toy* and *Tof* are described in Section 5.4.

5.5.3 Loop Interchange

For loop interchange, we first serially compute Loop-1 and then Loop-2 as described in Section 5.2. 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.

6. PROPOSED CNN ACCELERATOR

A hardware CNN accelerator with a new dataflow and PE architecture is proposed to implement the optimized acceleration scheme in Section 5.5 and minimize latency, on-/off-chip memory access, and data communication.

6.1 Convolution Dataflow

The dataflow of input pixels from on-chip buffers to local register arrays to PEs is shown in Figure 7. Unrolling Loop-3 provides us the opportunity to reuse overlapped input pixels by kernel window sliding. A register array is designed to reuse these data by data movements between registers similar to the dataflow in [6], which is realized by unrolling both Loop-1 and Loop-3.

The numbers in Figure 7 (e.g., 14) denote the (y, x) location of pixels (e.g., (y=1, x=4)) in the input feature map. In this example, *Pox* = 3, *Poy* = 3, and the number of input pixel buffer and register arrays equals *Poy*. Pixels of one tiling block (*Tix* = 8, *Tiy* = 6) are first stored in input pixel buffers row-by-row as shown in blue color. At clock cycle 0, pixels at address 0 are loaded into the corresponding registers as shown by blue dashed box to blue solid box with zero padding masked. With Nkx = Nky = 3 in this example, pixels are shifted within register arrays at cycle 1,2,4,5,7,8 as shown by green arrows, where sliding overlapped pixels are reused. At cycle 3 and 6, new pixels are transferred from Input Pixel BUF 1 and BUF 2 to the 3rd register array (R3*), respectively, and old pixels are shifted and reused across register arrays (R3*→R2*, R2*→R1*) as depicted by arrows from dashed yellow box to solid yellow boxes. For simplicity's sake, not every data movement is shown in Figure 7. After Nkx×Nky cycles, we complete one kernel window sliding (Loop-1) and move to the next input feature map with the same dataflow. After Nkx×Nky×Nif cycles, both Loop-1 and Loop-2 are completed and we obtain *Pox*×*Poy*×*Pof* final output pixels. The MAC units can continuously receive and compute input data from buffers without idle cycles. This dataflow is scalable to Nkx and Nky by only changing the control logic for sequential computing. As we judiciously do not unroll Loop-1, our dataflow has no partial sum movements between MAC units, and input pixels are also reused by *Pof* MAC units as shown in Figure 8 by unrolling Loop-4, both of which are not achieved in [6].

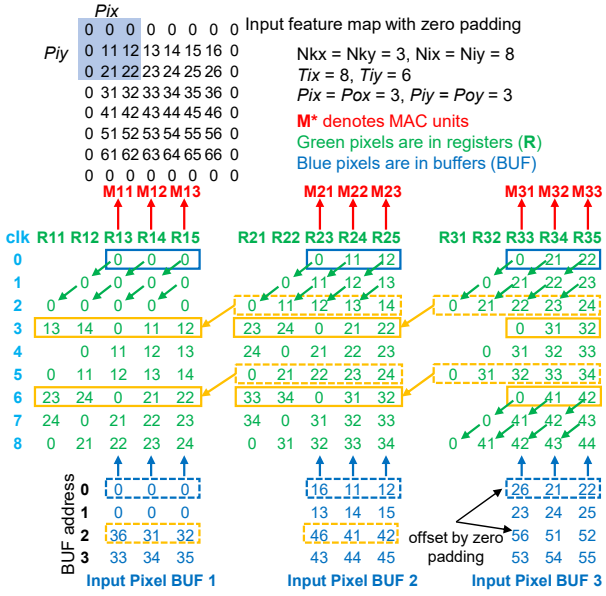

Figure 7. Dataflow of input pixels in convolution layers.

6.2 Convolution PE Architecture

The computing architecture of convolution layers is designed according to the proposed dataflow shown in Figure 8, which is comprised of *Pox*×*Poy*×*Pof* PEs and *Poy* input pixel register arrays. 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, 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 can also be reused within the register arrays. Every input pixel buffer is connected with its corresponding register array, except for the first two buffers that are also connected to the *Poy*-th register array determined by the dataflow. 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*.

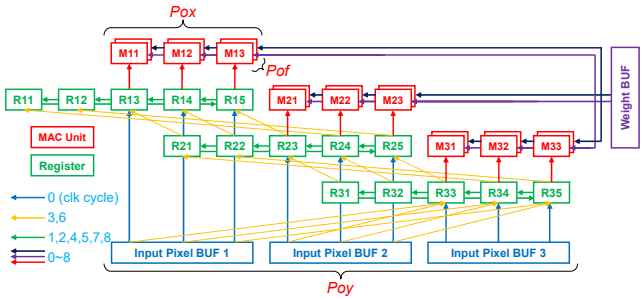

Figure 8. Convolution acceleration architecture.

After the completion of Loop-1 and Loop-2, the partial sums need to be added with biases as in Figure (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 Nkx×Nky×Nif > *Poy* 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, and the bandwidth of output pixel buffer can also be reduced by a factor of *Poy*. The bias adder results are truncated to be pixel_datawidth, and the location of the retained bits inside the bias adder results are moved according to the range of the data values in different layers, which allows dynamically adjusting the decimal points to achieve higher effective precision with the same data width. The rectified linear unit (ReLU) activation operation is performed after truncation with a 2-to-1 multiplexer, which uses the sign bit as the select signal and the truncated pixel and zero as the inputs. Finally, the outputs of ReLU are written into *Pof* output pixel buffers, where one buffer continuously stores *Pox*×*Poy* pixels on the same feature map. These processes after MAC compuations are not included in Figure 8.

6.3 Pooling Dataflow and PE Architecture

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. Figure 9 shows the dataflow and PE architecture of the pooling layers. The output pixels from previous convolution layers are stored row-by-row in the output pixel buffers. As pooling operation only need pixels, after one tiling of convolution 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 as the convolution layers, except for *Poy* = 1, as the width of output pixel buffer is only *Pox*. By this means, we can enable *Pox*×*Pof* parallel pooling operations, which is large enough considering much less operations in pooling layers compared to convolution layers. Four register arrays are used to reshape the pooling input pixels and hide the buffer read latency. At clock cycle 0 and 1, pixels at address 0 and 1 are loaded to the left and right part of both

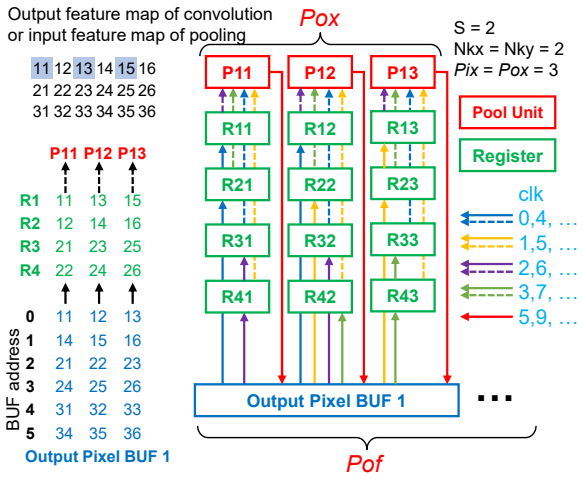

Figure 9. Pooling acceleration dataflow and architecture.

Table 2. Comparison with Previous CNN FPGA Implementations

	[13] AlexNet	[12] AlexNet	[11] AlexNet	[10] VGG	[9] VGG	[8] VGG	This work: VGG
FPGA	Virtex-7 VC 707	Virtex-7 VC709	Stratix-V GXA7	Zynq XC7Z045	Stratix-V GSD8	Virtex-7 VX690t	**Arria-10 GX 1150**
Frequency (MHz)	160	156	100	150	120	150	**150**
# Operations (GOP)	1.33	1.46	1.46	30.76	30.95	30.95	**30.95**
Number of Weights	2.33 M	60.95 M	60.95 M	50.18 M	138.3 M	138.3 M	**138.3 M**
Precision (all fixed)	32 bit	16 bit	8-16 bit	16 bit	8-16 bit	16 bit	**8-16 bit**
DSP Utilization	2,688 (96%)	2,144 (60%)	256 (100%)	780 (89%)	1,963[d]	3,600[d]	**1,518 (100%)**
Logic Utilization[a]	45K (9.2%)	274K (63%)	121K (52%)	183K (84%)	262K[d]	693K[d]	**161K (38%)**
On-chip RAM[b]	543 (53%)	956[b] (65%)	1,552 (61%)	486 (87%)	2,567[d]	1,470[d]	**1,900 (70%)**
Latency/Image (ms)	-	8×2.56[c]	12.75	224.6	262.9	151.8	**47.97**
Throughput (GOPS)	147.82	565.94	114.5	136.97	117.8	203.9	**645.25**

[a.] Xilinx FPGAs in LUTs and Altera FPGAs in ALMs
[b.] Xilinx FPGAs in BRAMs (36 Kb) and Altera 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.
[d.] The resource utilization is not reported in the original papers. The total available resources of the used FPGAs are listed.

the 1st (R1*) and 2nd (R2*) register arrays, respectively. The operations for 3rd (R3*) and 4th (R4*) register arrays are similar. Aided by this dual buffer technique, pixels can be continuously fed into PEs without idle cycles. The PEs for max-pooling are made of comparators. The outputs of pooling are written back to the output pixel buffers and then transferred to the external memory.

6.4 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 PE array used in convolution layers for all the fully-connected layers. Contrary to convolution layers, FC layers normally 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 FC weight buffers are used to overlap the inner-product computation with off-chip communication. FC layer output pixels are directly stored in on-chip buffers as their size is small (<20 KB).

7. EXPERIMENTAL RESULTS

7.1 System Setup

The proposed hardware CNN accelerator is demonstrated by implementing VGG-16 CNN model [3], which has 13 convolution layers, 5 pooling layers, 3 FC layers and 138.3 million parameters. We used a Nallatech 385A board that includes two banks of 4GB DDR3L SDRAM and an Altera Arria 10 GX 1150 FPGA, which consists of 1,150K logic elements, 1,518 DSP blocks and 2,713 M20K memory blocks.

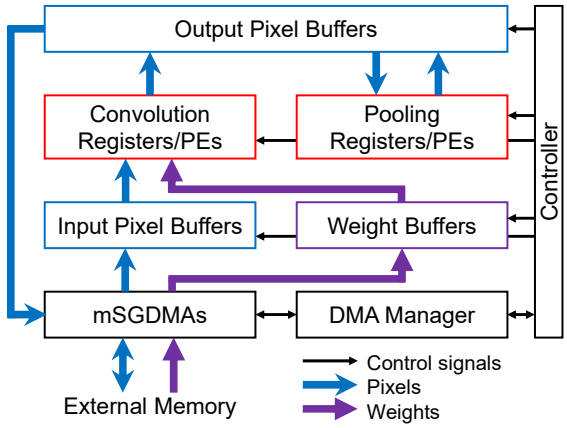

Figure 10. Top-level CNN acceleration system.

The overall CNN acceleration system on the FPGA chip shown in Figure 10 is coded in parametrized Verilog scripts and compiled by Quartus Prime synthesis tool. With two SDRAM banks, both kernel and feature maps are separated into these two banks to enable full off-chip communication. Two Modular Scatter-Gather DMA (mSGDMA) engines provided by Altera are used to simultaneously read and write from/to these two SDRAM banks. After the input images and weights are loaded into SDRAMs, the CNN acceleration process starts. When the computation of one tiling loop completes, the output pixels are transferred to SDRAM, and then the weights and pixels for the next tiling loop are loaded from SDRAM 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.

The fixed point data representation is used with 16-bit pixels, 8-bit weights, and 30-bit partial sums. The decimal points can be dynamically adjusted according to the ranges of pixel values in different layers to fully utilize the existing data width [10][11]. By this means, the top-1 and top-5 ImageNet classification accuracy degradation is within 2% compared with software full precision implementation [8][9][10].

7.2 Analysis of Experimental Results

The performance and specifications of the CNN accelerator is summarized and compared with other CNN FPGA implementations in Table 2. In Arria 10 FPGA, one DSP block can support two 18-bit×18-bit multiplications and our Arria 10 FPGA has 1,518 DSP blocks, which can support 3,036 multiplications in parallel. Our unrolling strategy needs $Pm = Pox \times Poy \times Pof = 3{,}136$ multiplications, and the remaining 100 multipliers are implemented by logic elements. The breakdown of the processing time per image of VGG-16 model is shown in Figure 11. The computation time of convolution layers dominates the total latency by 70.0%. DMA_conv, which consumes 10.1% of the overall latency, includes the SDRAM transaction delay of convolution weights and input/output pixels. The FC computing time is hidden by FC weights transfer delay through DMA, shown as DMA_FC.

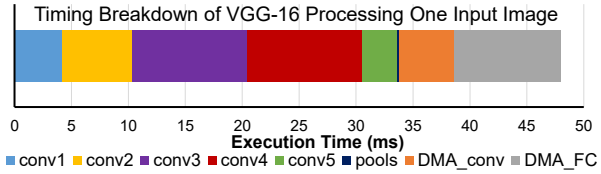

Figure 11. Latency breakdown of VGG-16 acceleration.

The breakdown of our on-chip RAM (total of 1,900 M20K blocks) utilization is: convolution pixel/weight buffers (70.1%), FC buffers (11.9%), FIFOs for clock crossing (3.1%) and DMA (11.7%). The actual M20K bits (27.2 Mbits) used for convolution buffers is larger than the optimized results of 20.8 Mbits, because the required memory depth value is not a power of 2.

The power of 385A FPGA card can only be supplied through a PCIe slot, which makes it difficult to directly measure the power consumption. Instead, we use Altera PowerPlay Power Analyzer to simulate our power consumption. The total thermal power is 21.2 W consisting of sources from DSP (3.6%), M20K (15.0%), logic cells (16%), clock (12.9%), transceiver (15.7%), I/O (13.2%) and static consumption (23.5%).

7.3 Comparison with Prior Works

The reported results from recent CNN FPGA accelerators are listed in Table 2. [13] only implements convolution layers in AlexNet and uses the similar strategy as us to unroll Loop-3 and Loop-4, which can also achieve high DSP utilization. However, their 32-bit data width forces the multipliers to use much more DSP blocks than our design. In addition, their tiling strategy is only along Loop-3 and Loop-4, which significantly postpones the acquisition of the final pixels resulting in more memory accesses and data movements of partial sums.

In [8][12], the layer-by-layer computation is pipelined using different part of one or multiple FPGAs resources. Pipelining of each layer may increase hardware utilization thus achieve high throughput by preventing computing engines from being idle caused by the imbalance of different layers. However, with the highly increasing number of convolution layers [4], 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. On the contrary, our uniform architecture can also be fully utilized by all the convolution layers and is highly scalable to be applied to other CNNs with more layers, e.g. the ResidualNet [4].

Batch computing with multiple input images is applied in [6][8][12]. 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. As the DRAM transaction delay of kernel weights accounts for a signification portion (~22%) in our system latency as shown in Figure 11, batch computing could also benefit our throughput considerably by effectively hiding the DMA weight transfer delay. Benefit from batch computing and using 2,144 DSP blocks, which enables high parallelism degree, [12] also achieves high throughput of 565.94 GOPS for AlexNet.

With the optimized CNN acceleration scheme and low-communication dataflow, the proposed CNN accelerator uses uniform unrolling factors for all the convolution layers and fully utilizes all the DSP resources to achieve 3.2–5.6× throughput improvement than [8][9][10][11][13] and 3.2–5.5× less latency than prior VGG implementations [8][9][10].

8. 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, and we provide design guidelines for an efficient acceleration strategy. A corresponding new dataflow and architecture is proposed to minimize data communication and enhance throughput. Our CNN accelerator demonstrated VGG-16 CNN model on Arria 10 FPGA, achieving 645.25 GOPS of throughput and 47.97 ms of latency per image, which outperforms all prior VGG FPGA implementations by >3.2×.

9. ACKNOWLEDGMENT

This work was supported in part by the NSF I/UCRC Center for Embedded Systems through National Science Foundation grant 1361926 and 1535669, and Samsung Advanced Institute of Technology.

10. REFERENCES

[1] O. Russakovsky, et al. ImageNet large-scale visual recognition challenge. In *Int. J. Computer Vision*, 2015.

[2] A. Krizhevsky, et al. ImageNet classification with deep convolutional neural networks. In *Neural Information Processing Systems (NIPS)*, 1097-1105, 2012.

[3] K. Simonyan, et al. Very Deep Convolutional Networks for Large-Scale Image Recognition. In *Int. Conf. Learning Representations (ICLR)*, 2015.

[4] K. He, et al. Deep residual learning for image recognition. In *IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[5] Y.-H. Chen, et al. Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks. In *IEEE Int. Solid-State Circuits Conf. (ISSCC)*, 2016.

[6] Y.-H. Chen, et al. Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks", In *ACM/IEEE Int. Symp. Computer Architecture (ISCA)*, 2016.

[7] C. Zhang, et al. Optimizing FPGA-based accelerator design for deep convolutional neural networks. In *ACM Int. Symp. on Field-Programmable Gate Arrays (FPGA)*, 2015.

[8] C. Zhang, et al. Energy-Efficient CNN Implementation on a Deeply Pipelined FPGA Cluster. In *ACM Int. Symp. on Low Power Electronics and Design (ISLPED)*, 326-331, 2016.

[9] N. Suda, et al. Throughput-optimized OpenCL-based FPGA accelerator for large-scale convolutional neural networks. In *Int. Symp. on Field-Programmable Gate Arrays (FPGA)*, 16-25, 2016.

[10] J. Qiu, et al. Going deeper with embedded FPGA platform for convolutional neural network. In *ACM Int. Symp. on Field-Programmable Gate Arrays (FPGA)*, 26-35, 2016.

[11] Y. Ma, et al. Scalable and Modularized RTL Compilation of Convolutional Neural Networks onto FPGA, In *Int. Conf. Field-Programmable Logic and Applications (FPL)*, 2016.

[12] H. Li, et al. A High Performance FPGA-based Accelerator for Large-Scale Convolutional Neural Networks, In *Int. Conf. Field-Programmable Logic and Applications (FPL)*, 2016.

[13] A. Rahman, et al. Efficient FPGA acceleration of Convolutional Neural Networks using logical-3D compute array. In *Design, Auto. & Test in Europe Conf. (DATE)*, 2016.

[14] M. Motamedi, et al. Design space exploration of FPGA-based Deep Convolutional Neural Networks. In *Asia and South Pacific Design Auto. Conf. (ASP-DAC)*, 2016.

[15] S. Han, et al. Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding. In *Int. Conf. Learning Representations (ICLR)*, 2016.

[16] S. Han, et al. EIE: Efficient Inference Engine on Compressed Deep Neural Network. In *ACM/IEEE Int. Symp. Computer Architecture (ISCA)*, 2016.

[17] D. Bacon, et al. Compiler transformations for high-performance computing. In *ACM Computing Surveys (CSUR)*, Volume 26 Issue 4, Pages 345-420, 1994.