Performance Modeling for CNN Inference Accelerators on FPGA

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

***Abstract*—The recently reported successes of convolutional neural networks (CNNs) in many areas have generated wide interest in the development of field-programmable gate array (FPGA)-based accelerators. To achieve high performance and energy efficiency, an FPGA-based accelerator must fully utilize the limited computation resources and minimize the data communication and memory access, both of which are impacted and constrained by a variety of design parameters, e.g., the degree and dimension of parallelism, the size of on-chip buffers, the bandwidth of the external memory, and many more. The large design space of the accelerator makes it impractical to search for the optimal design in the implementation phase. To address this problem, a performance model is described to estimate the performance and resource utilization of an FPGA implementation. By this means, the performance bottleneck and design bound can be identified and the optimal design option can be explored early in the design phase. The proposed performance model is validated using a variety of CNN algorithms comparing the results with on-board test results on two different FPGAs.**

***Index Terms*—Analytical modeling, convolutional neural networks (CNNs), field-programmable gate array (FPGA).**

I. Introduction

MANY reported successes of convolutional neural networks (CNNs) for computer vision tasks [1]–[6] have motivated the development of hardware implementations of CNNs. In particular, there has been increased interest in field-programmable gate arrays (FPGAs) as a platform to accelerate the post-training inference computations of CNNs [7]–[13]. To achieve high performance and low energy cost, a CNN accelerator must: 1) fully utilize the limited computing resources to maximize the parallelism when executing the large number of operations for different convolution layers with varying dimensions; 2) exploit the data locality by saving only the required data in on-chip buffers to minimize the cost of external memory (e.g., DRAM) accesses; and 3) manage the data storage patterns in buffers to increase the data reuse and reduce the data movements.

With the intervals of computation and off-chip communication overlapped using the dual buffering (or ping-pong buffering) technique, the performance of the CNN accelerator will be limited by either the computation delay or the DRAM transfer delay, and the actual bound will be determined by the values of the associated design parameters, as described by the *roofline model* in [7] and [9]. The computation delay is determined by the number of parallel processing engines (PEs), their utilization, and the operating frequency. The DRAM transfer latency is mainly affected by the external memory bandwidth and the number of DRAM accesses, and the latter is strongly affected by the size of the on-chip buffers. With regard to the energy efficiency (i.e., performance per watt), the main components that determine the dynamic power consumption are the computation logic and the memory traffic, the latter requiring efficient data movement and high data reuse. All these considerations show that there are numerous design parameters that determine the performance and energy efficiency of a CNN accelerator, making it impractical to find their optimal values during the implementation phase, as the synthesis of one FPGA design may take several hours. Robust and parametric models become a necessity for efficient design space exploration and selection of the optimal values of the design parameters. The architectural design space must be numerically characterized by design variables to control the accelerator performance and efficiency. For instance, loop optimization techniques [7], [11], such as loop unrolling and tiling, are employed to customize the acceleration strategy of parallel computation and data communication for convolution loops, whose variables in turn affect the resource utilization and memory access.

The starting point of this paper is a general system-level model of a CNN accelerator shown in Fig. 1, which includes the external memory, on-chip buffers, and PEs. The hardware architectural parameters, e.g., buffer sizes, are determined by the design variables that control the loop unrolling and tiling. Combining the design constraints and the choices of the acceleration strategy, a more fine-grained performance model is built to achieve better prediction for a specific design implementation, e.g., the design strategy in [11]. By this means,

Manuscript received August 27, 2018; revised December 6, 2018; accepted January 17, 2019. Date of publication February 4, 2019; date of current version March 18, 2020. 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, in part by the Intel Labs, and in part by the Center for Brain-Inspired Computing (C-BRIC), one of six centers in JUMP, an SRC program sponsored by DARPA. This paper was recommended by Associate Editor Y. Wang. *(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).

Digital Object Identifier 10.1109/TCAD.2019.2897634

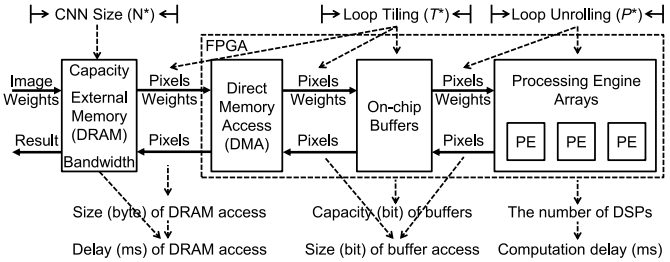


Fig. 1. General CNN hardware accelerator with three levels of hierarchy, where the loop design variables determine the key accelerator metrics, e.g., delay, resource usage, and memory access.

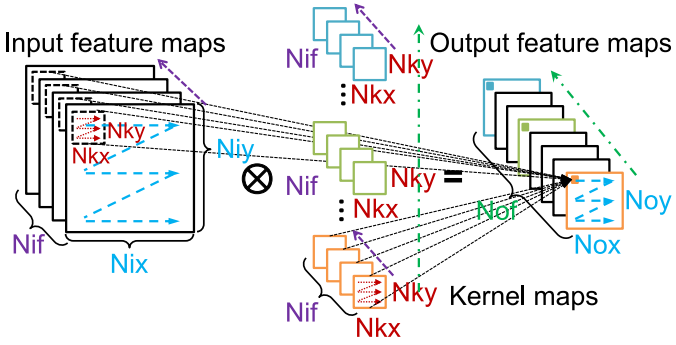


Fig. 2. Convolution operation is implemented by four levels of loops to MAC input features with kernel weights, where *i*: input, *o*: output, *k*: kernel, *f*: feature, *x*: *x*-axis (width), and *y*: *y*-axis (height), and the parameters of loop dimensions are prefixed with *N* [11].

the proposed performance model makes it possible to identify the performance bottleneck and design limitations in the early development phase by exploring the design space through unrolling and tiling variables.

The main contributions of this paper are as follows.

1) The design objectives and resource costs are formulated using the design variables of loop unrolling and tiling.
2) A high-level performance model is proposed to estimate the accelerator throughput, on-chip buffer size, and the number of external and on-chip memory accesses.
3) The design space is efficiently explored through the proposed model instead of the real FPGA compilation to identify the performance bottleneck and obtain the optimal design configurations.
4) The performance model is validated for a specific design strategy across a variety of CNN algorithms comparing with the on-board test results on two different FPGAs.
5) Evaluate the techniques that may further enhance the performance of our current design by improving the efficiency of DRAM transactions and PE utilization.

The remainder of this paper is organized as follows. Section II overviews the procedure to map CNN operations onto an FPGA hardware system. A coarse-grained performance model is presented in Section III for rough estimation, and the fine-grained model is discussed in the following sections for a specific design strategy. Section IV estimates the size and latency of DRAM accesses. The latency of convolution and other layers are formulated and estimated in Section V. The on-chip buffer size requirement is analyzed in Section VI, and the size of buffer access is discussed in Section VII. Experiments are performed to explore the design space in Section VIII. Section IX evaluates the techniques that may further improve the current design performance.

II. CNN Inference Accelerator on FPGA

A. Overview of Convolution Operation

The main operation in CNN algorithms involves accumulating the products of pixel values (e.g., features, activations, or neurons) with kernel weights, along different dimensions of the kernel and feature maps. Fig. 2 shows the four nested loops involved in CNNs. Note that the prefix *N* (for "number") used in describing various parameters (e.g., *Nix*, *Niy*, *Nif*, etc.) denote the sizes of the kernel and feature maps. The loop operations shown in Fig. 2 are written as

$$\text{pixel}_L(\text{no}; x, y) = \sum_{ni=1}^{Nif} \sum_{ky=1}^{Nky} \sum_{kx=1}^{Nkx} \text{pixel}_{L-1}(ni; \text{S} \times x + kx, \text{S} \times y + ky) \times \text{weight}(ni, \text{no}; kx, ky) + \text{bias}(\text{no}) \quad (1)$$

where S is the sliding stride, $x \in \{1, 2, \ldots, Nox\}$, $y \in \{1, 2, \ldots, Noy\}$, $\text{no} \in \{1, 2, \ldots, Nof\}$, $L \in \{1, 2, \ldots, \#\text{CONVs}\}$, and #CONVs is the number of convolution layers.

B. CNN Hardware Acceleration System

In the general model of a CNN accelerator shown in Fig. 1, due to the large data volume, both the weights and intermediate pixel results are stored in the external memory, e.g., DRAM. The input and weight on-chip buffers temporally store the input data to be processed by the PEs, and the PE results are saved in the output buffers. After completing the computation, the results are transferred back to DRAM from the output buffers, which will be used as the input to the subsequent layer.

C. Convolution Loop Optimization

Loop optimization techniques [7], [11], e.g., unrolling and tilling, are employed to customize the computation and communication patterns in a CNN accelerator. Loop unrolling directs the parallel computation along different convolution dimensions, and the variables representing the unrolling degrees are prefixed by *P* (see Fig. 3). These variables determine the number of PEs, which in turn determine the required number of DSPs in the FPGA to implement PEs, and thus decide the computation delay. The data flow from buffers into PEs is also impacted by loop unrolling variables, which affect the number of buffer access. Loop tilling divides a large CNN layer into multiple small tiles, which can be accommodated by on-chip buffers to increase data locality. Tiling sizes are represented by variables prefixed with *T* as shown in Fig. 3. The required buffer capacity is determined by the tiling variables, which also affect the DRAM access and thus the latency of DRAM transactions. The relationship between loop variables and the key specifications of accelerators, e.g., delay, DSP usage, buffer size, and memory access that affect memory power consumption, are shown in Fig. 1.

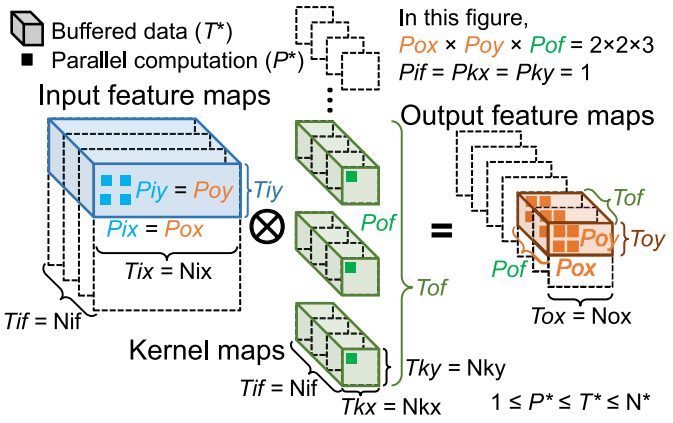


Fig. 3. Convolution acceleration strategy is customized by loop unrolling ($P*$) for parallel computation and loop tiling ($T*$) for data buffering. The parallelism is within one input feature map ($Pix \times Piy$) and across multiple kernel maps (Pof). The demanded buffer size can be changed by tuning variables Toy and Tof [11].

D. Convolution Acceleration Strategy

To accurately predict the real implementation, a specific accelerator design strategy is needed to characterize the fine-grained performance model with detailed design options. The output stationary acceleration strategy of unrolling and tilling in [11] is adopted in this paper as shown in Fig. 3. The loop unrolling or the parallel computations are only performed within one input feature map ($Pix = Pox > 1$ and $Piy = Poy > 1$) and across multiple kernel maps ($Pof > 1$). That is, in Fig. 3, the $Pix \times Piy$ blue pixels in an input feature map are operated in parallel with green weights from Pof different kernel maps, resulting in $Pox \times Poy$ pixels in each of the Pof output feature maps. Therefore, the total number of PEs [multiply and accumulate (MAC) units] is $Pox \times Poy \times Pof$. The data required to compute one final output pixel are fully buffered, i.e., $Tkx = Nkx$, $Tky = Nky$, and $Tif = Nif$, so that the partial sum can be consumed inside the MAC unit without saving in the buffer. To ensure that the DRAM accesses are from continuous addresses, the entire row of the feature map is buffered, i.e., $Tix = Nix$ and $Tox = Nox$. If the on-chip RAM capacity is large enough, either all pixels or all weights of one layer are fully buffered, so that each data only needs to be fetched from DRAM once to reduce DRAM access. Finally, the required buffer sizes of each layer can be changed by tuning Toy and Tof.

III. Coarse-Grained Performance Model

In this section, a coarse-grained performance model of a general CNN accelerator that is independent of a specific acceleration strategy, is presented. Then, more detailed design choices and constraints (e.g., unrolling and tiling variable settings, memory storage pattern, and computation dataflow) are introduced to create a more precise and fine-grained model in the following sections. Table I lists the mainly used abbreviations and units in this paper, which indicate the meaning of the variables discussed afterward.

TABLE I
List of Abbreviations and Units

Abbreviation	Description	Abbreviation	Description
Px	Pixel	Rd	Read
Wt	Weight	Wr	Write
Buf	Buffer	$InBuf$	Input Buffer
$WtBuf$	Weight Buffer	$OutBuf$	Output Buffer
BW	Bandwidth	$1T$	One Tile
Unit	Description	Unit	Description
bit / $byte$	Data Size	$word$	RAM Depth
ms	Delay Time	MHz	Frequency

A. Computation Latency

The number of multiplication operations per layer is $\mathrm{Nm} = Nif \times Nkx \times Nky \times Nof \times Nox \times Noy$. The number of PEs that determines the degree of parallel computations by unrolling is $Pm = Pif \times Pkx \times Pky \times Pof \times Pox \times Poy$. A similar reasoning is applied to determine the number of clock cycles for one buffered tile ($1T$) of convolution. This is denoted by $\#cycles\_1T$, and is expressed as follows:

$$\#cycles\_1T = \left\lceil \frac{Tif}{Pif} \right\rceil \left\lceil \frac{Tkx}{Pkx} \right\rceil \left\lceil \frac{Tky}{Pky} \right\rceil \left\lceil \frac{Tof}{Pof} \right\rceil \left\lceil \frac{Tox}{Pox} \right\rceil \left\lceil \frac{Toy}{Poy} \right\rceil. \tag{2}$$

The number of tiles for one convolution layer is

$$\#tiles = \left\lceil \frac{Nif}{Tif} \right\rceil \left\lceil \frac{Nkx}{Tkx} \right\rceil \left\lceil \frac{Nky}{Tky} \right\rceil \left\lceil \frac{Nof}{Tof} \right\rceil \left\lceil \frac{Nox}{Tox} \right\rceil \left\lceil \frac{Noy}{Toy} \right\rceil. \tag{3}$$

The total number of computation clock cycles of one convolution (CV) layer is

$$\#cycles\_1CV = \#tiles \times \#cycles\_1T. \tag{4}$$

B. On-Chip Buffer Size

Determined by the tiling variables, the input buffer ($InBuf$) size (bit) requirement to store one tile of input pixels is

$$bit\_InBuf = Tix \times Tiy \times Tif \times bit\_Px \tag{5}$$

where bit_Px is the bit width of one pixel (Px). Similarly, the size (bit) requirement of weight buffer ($WtBuf$) to store one tile of weights is

$$bit\_WtBuf = Tkx \cdot Tky \cdot Tif \cdot Tof \cdot bit\_Wt \tag{6}$$

where bit_Wt is the bit width of one weight (Wt). The output buffer ($OutBuf$) size (bit) requirement to store one tile of output pixels is

$$bit\_OutBuf = Tox \times Toy \times Tof \times bit\_Px. \tag{7}$$

The theoretical sizes of the input, weight, and output buffers are the maximum possible values of bit_InBuf, bit_WtBuf, and bit_OutBuf of all the convolution layers, respectively. In an actual implementation, the sizes of the buffers used may be larger than these values due to inefficient storage pattern and extra garbage data.

C. DRAM Access and Latency

In theory, the size of one tile of data read from or written to the external DRAM should be the same as the size of buffered data. Therefore, the size (bytes) of input pixels (*Px*) read (*Rd*) from DRAM for one convolution tile is $byte\_RdPx = bit\_InBuf/8$. The size (bytes) of one tile of weights (*Wt*) read from the DRAM is $byte\_RdWt = bit\_WtBuf/8$. The size (bytes) of one tile of output pixels written (*Wr*) to the DRAM is $byte\_WrPx = bit\_OutBuf/8$. The latency (milliseconds or ms) of DRAM transactions of one tile ($1T$) of data is determined by the size of DRAM access and the memory bandwidth. This is given by

$$ms\_DRAM\_1T = \frac{byte\_DRAM\_1T}{BW\_Memory \times 10^6} \tag{8}$$

where *BW_Memory* is the external memory bandwidth (GB/s) and $byte\_DRAM\_1T$ is the size of DRAM access of one tile, which can be *byte_RdPx*, *byte_RdWt*, or *byte_WrPx*.

D. On-Chip Buffer Access

The size (bits) of on-chip buffer access (*bit_Buf_Access*) is computed by multiplying the number of access clock cycles (*#cycles_Access*) with the total bit width of the corresponding buffers (*width_Buf*)

$$bit\_Buf\_Access = \#cycles\_Access \times width\_Buf. \tag{9}$$

During computation, it is assumed that data are continuously read from input and weight buffers and the results are written into the output buffers every clock cycle. Then, to estimate the buffer access during computation, *#cycles_Access* equals the number of computation cycles, and *width_Buf* can be the total bit width of input/weight/output buffers. The size (bits) of buffer access by direct memory access (DMA) that writes into input and weight buffers and reads from output buffers is the same as the size of external memory access. The data stored in the input or weight buffers may be read multiple times during computation, hence the size of data read from buffers may be larger than the size of data written into buffers from DRAM. Since each result is written into output buffers only once, the size of write and read operations of output buffers is the same.

E. Other Implementation Methods of Convolution

Instead of the aforementioned direct implementation of the convolution loop operations, convolution can also be performed as matrix multiplication [8] or accelerated in the frequency domain [14], [15]. Since these methods require significantly different hardware architecture and dataflow, we only briefly analyze them with our modeling parameters.

1) Matrix Multiplication: The MAC operations in convolution can be mapped to matrix multiplication [8], which can utilize the library optimized for GPU, e.g., BLAS used by Caffe. The original 4-D kernel weights are transformed to be a matrix with Nof rows and $Nkx \cdot Nky \cdot Nif$ columns. The 3-D input feature map is transformed into a matrix with $Nkx \cdot Nky \cdot Nif$ rows and $Nox \cdot Noy$ columns. There are redundant data in the transformed feature matrix due to the overlapped sliding of the kernel window. Therefore, this method could lead to either complex dataflow and extra hardware to perform the transform on the fly or additional DRAM accesses due to the redundant data.

2) Fast Fourier Transform: FFT [16] can reduce the number of multiplications from $\Theta(Nox \cdot Noy \cdot Nkx \cdot Nky)$ to $\Theta(Nox \cdot Noy \cdot \log_2(Nox))$ and even further to $\Theta(Nox \cdot Noy \cdot \log_2(Nkx))$ with Overlap-and-Add [16]. The original kernel weights and input features are transformed into the frequency domain to do multiplications, and then the inverse FFT is applied to recover the results. Therefore, extra hardware is required to implement the transform. In addition, the computation reduction is decreased with smaller kernel size.

3) Winograd Transform: Winograd transform [15] is another approach to reduce the convolution operations. The number of multiplications of the original convolution in one tile is $\Theta((Tkx \cdot Tox)(Tky \cdot Toy))$, while in Winograd it is only $\Theta((Tkx + Tox - 1)(Tky + Toy - 1))$. For example, with kernel size of 3×3 and tiled output features of 2×2, we can achieve $2.25\times$ reduction of multiplication operations. However, the addition operations are increased in Winograd, and additional storage and bandwidth are required by the transform matrices. The operations can be further reduced with larger feature tiles, but the complexity of the transform matrix will significantly increase. Since Winograd essentially unrolls the computation within a kernel window, the varying kernel sizes can affect its computation efficiency.

IV. Modeling of DRAM Access

In this section, more accurate models of the DRAM access are constructed by including the design constraints and the variables of loop acceleration described in Section II-D.

A. Data Size of Convolution DRAM Access

The DMA engine shown in Fig. 1 is used to transfer data to and from off-chip DRAM. To achieve the maximum bandwidth, the data width of both the DMA (*bit_DMA*) and the DRAM controller (*bit_DRAM*) are set to be 512 bits.

Pox represents the number of pixels that are computed in parallel in each output feature map. For the feature map transfer, the number of groups of *Pox* pixels associated with one DMA address is then given by $\#PoxGroup = \lfloor bit\_DMA/(Pox \times bit\_Px) \rfloor$, where *bit_Px* is the bit width per pixel. The effective or actual DMA bandwidth (as a fraction of the maximum) is then given by

$$eff\_DMA\_Px = \frac{\#PoxGroup \times Pox \times bit\_Px}{bit\_DMA}. \tag{10}$$

For example, if $Pox = 7$, $bit\_DMA = 512$, and $bit\_Px = 16$, then there are $\#PoxGroup = 4$ groups of *Pox* pixels in one DMA address, and $4 \times 7 \times 16 = 448$ bits are the effective number of bits out of the DMA bit width of 512 bits, resulting in $eff\_DMA\_Px = 0.875$.

The intermediate pixel results stored in DRAM are arranged row-by-row, map-by-map, and layer-by-layer. One convolution tile needs $Tix \times Tiy \times Tif$ input pixels. Then, the size (bytes)

of the input pixels read (*Rd*) from the DRAM for one tile is

$$byte\_RdPx = \frac{Tix \times Tiy \times Tif \times bit\_Px}{eff\_DMA\_Px \times 8}. \quad (11)$$

Note that if $eff\_DMA\_Px < 1$, it implies more bytes are read than necessary due to the alignment of data storage. Similarly, the size (bytes) of output pixels written (*Wr*) to DRAM for one convolution tile is

$$byte\_WrPx = \frac{Tox \times Toy \times Tof \times bit\_Px}{eff\_DMA\_Px \times 8}. \quad (12)$$

For convolution weights, the ratio of effective DRAM bandwidth to the maximum of reading weights from DRAM is

$$eff\_DMA\_Wt = \frac{\lfloor bit\_DMA/bit\_Wt \rfloor \times bit\_Wt}{bit\_DMA}. \quad (13)$$

The size (bytes) of input weights read from DRAM for one convolution tile is

$$byte\_RdWt = \frac{Tkx \cdot Tky \cdot Tif \cdot Tof \cdot bit\_Wt}{eff\_DMA\_Wt \times 8}. \quad (14)$$

B. DRAM Access Delay of One Tile

The data width of the DRAM controller interface to the FPGA is assumed to be *bit_DRAM*, running at frequency of *MHz_DRAM*. This means the theoretical maximum DRAM bandwidth (*BW_DRAM* in GB/s) is $(bit\_DRAM/8) \times (MHz\_DRAM/10^3)$, which is normally very difficult to sustain due to the noncontiguous DRAM access. For example, if $bit\_DRAM = 512$ bits, with $MHz\_DRAM = 266$ MHz, then $BW\_DRAM = (512/8) \times (266/10^3) = 17.0$ GB/s as the maximum DRAM bandwidth.

In the CNN acceleration system described in [11], the DMA engine is operated at the same clock frequency as the CNN accelerator core (i.e., *MHz_Accelerator*) with read/write data width (*bit_DMA*) of 512 bits. An asynchronous first-input, first-ouput (FIFO) can be inserted between DMA and the DRAM controller to synchronize data across the two clock domains. Then, the DMA bandwidth (*BW_DMA*) is $(bit\_DMA/8) \times (MHz\_Accelerator/10^3)$. By this means, the bandwidth of the external memory is bounded by the effective bandwidth of both the DRAM controller and the DMA as $BW\_Memory = \min(BW\_DRAM, BW\_DMA)$, which is used in (8) to calculate the DRAM latency.

The more accurate and specific DRAM access sizes of one tile (*byte_DRAM_1T*) are discussed in this section, including *byte_RdPx*, *byte_WrPx*, and *byte_RdWt*. Then, we can use (8) to compute their corresponding DRAM access delay (*ms_DRAM_1T*), e.g., *ms_RdPx*, *ms_WrPx*, and *ms_RdWt*, respectively.

C. DRAM Access of Other Layers

The DRAM access and performance of other layers, e.g., max pooling, fully connected (FC), and Eltwise, are also investigated and included in our performance model. Since the analysis process of theses layers are similar to the convolution layer, for simplicity, their detailed formulas used in the performance model are not presented.

The pixels of max-pooling layers are also transferred to and from the DRAM with loop tiling performed, depending on the adopted design choices [11], [17]. For max pooling, the calculation of the DRAM transfer sizes of input and output pixels are similar to *byte_RdPx* in (11) and *byte_WrPx* in (12), respectively.

The weights of FC layers are stored in DRAM in the same way as convolution and reuse the same weight buffers. Since the intermediate results of FC layers are small (<20 KB), they are always kept in the on-chip RAMs.

The Eltwise layer performs element-wise summation of the outputs of two layers. The Eltwise layer is executed after its two precedent layers are finished, so that it can directly read the results of one layer from the output buffers, without accessing DRAM. However, the Eltwise layer still needs to read the outputs of the other layer from DRAM as the output buffers were already refreshed.

V. Modeling of Latency

A. Computation Delay (ms) of One Convolution Tile

Setting $Pif = Pkx = Pky = 1$, $Tif = Nif$, $Tkx = Nky$, $Tkx = Nky$, and $Tox = Nox$ as described in Section II-D, (2) can be written as

$$\#cycles\_1T = Nif \cdot Nkx \cdot Nky \cdot \left\lceil \frac{Tof}{Pof} \right\rceil \cdot \left\lceil \frac{Nox}{Pox} \right\rceil \cdot \left\lceil \frac{Toy}{Poy} \right\rceil. \quad (15)$$

Then, the computation delay (ms) of one convolution tile is

$$ms\_Compute = \frac{\#cycles\_1T}{MHz\_Accelerator \times 10^3} \quad (16)$$

where *MHz_Accelerator* is the clock frequency of the accelerator in MHz. The number of tiles of one convolution layer (*#tiles*) is $\lceil Nof/Tof \rceil \lceil Noy/Toy \rceil$ based on (3) with $Nif = Tif$, $Nkx = Tkx$, $Nky = Tky$, and $Nox = Tox$ as described in Section II-D.

B. Overall Delay (ms) of One Convolution Layer

With dual buffering technique, the DRAM access is overlapped with computation to improve performance [7], [10]. The overall tile-by-tile delay of one convolution layer is illustrated in Fig. 4. Since the dual buffering pipeline is only within one layer with the current design choice, after the start of one layer and before the computation of the first tile, both the input pixels and weights (*Wt*) of one tile are first read from DRAM. This is shown as "Input + *Wt*" at the beginning of one layer in Fig. 4. Similarly, after the completion of the last tile's computation, its output pixels are transferred back into DRAM, which is shown as "Output" at the end in Fig. 4. Therefore, for each convolution layer, the delay of transferring inputs of the first tile and outputs of the last tile cannot be overlapped with the computation, and this delay is denoted as

$$ms\_Mem = ms\_RdPx + ms\_RdWt + ms\_WrPx. \quad (17)$$

If the convolution layer has only one tile that is $Tiy = Niy$ and $Tof = Nof$, there is no overlapping of memory transfer and computation as shown in Fig. 4(a), and the delay of this tile

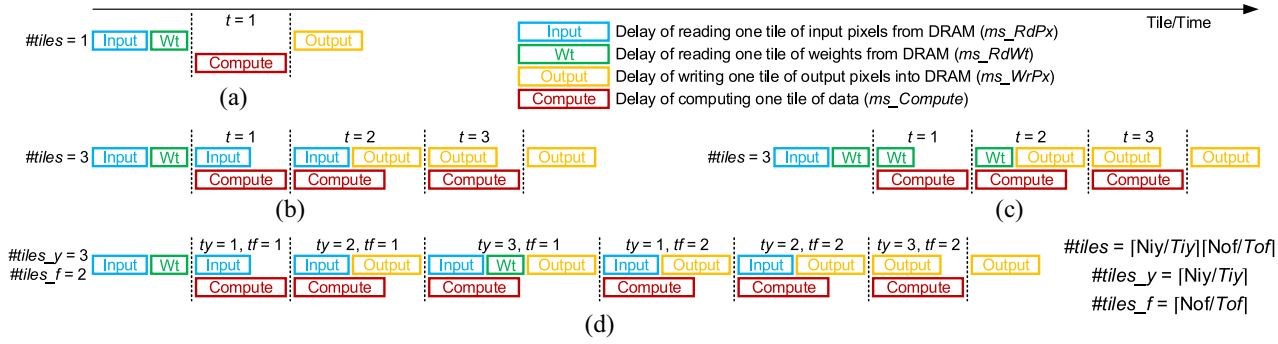


Fig. 4. Tile-by-tile delay of one convolution layer, and the DRAM access delay is overlapped with the computation delay due to dual buffering technique. (a) Both inputs and weights fully buffered, (b) only weights fully buffered, (c) only inputs fully buffered, and (d) neither inputs nor weights fully buffered.

Algorithm 1: Delay Estimation of one Convolution Layer (*ms_1CV*), Where $C = ms\_Compute$, $I = ms\_RdPx$, $W = ms\_RdWt$, and $O = ms\_WrPx$

input : C, I, W, O, $\#tiles$, $\#tiles\_y$, $\#tiles\_f$
output: ms_1CV

1 **if** $Tiy = \text{Niy}$ and $Tof = \text{Nof}$ **then**
2 $\quad T[1] = C$
3 **else if** $Tiy < \text{Niy}$ and $Tof = \text{Nof}$ **then**
4 $\quad$ **for** $t = 1$ **to** $\#tiles$ **do**
5 $\quad\quad$ **if** $t = 1$ **then**
6 $\quad\quad\quad T[t] = \max(C, I)$
7 $\quad\quad$ **else if** $t = \#tiles$ **then**
8 $\quad\quad\quad T[t] = \max(C, O)$
9 $\quad\quad$ **else**
10 $\quad\quad\quad T[t] = \max(C, I + O)$
11 $\quad\quad$ **end**
12 $\quad$ **end**
13 **else if** $Tiy = \text{Niy}$ and $Tof < \text{Nof}$ **then**
14 $\quad$ **for** $t = 1$ **to** $\#tiles$ **do**
15 $\quad\quad$ **if** $t = 1$ **then**
16 $\quad\quad\quad T[t] = \max(C, W)$
17 $\quad\quad$ **else if** $t = \#tiles$ **then**
18 $\quad\quad\quad T[t] = \max(C, O)$
19 $\quad\quad$ **else**
20 $\quad\quad\quad T[t] = \max(C, W + O)$
21 $\quad\quad$ **end**
22 $\quad$ **end**
23 **else**
24 $\quad$ **for** $tf = 1$ **to** $\#tiles\_f$ **do**
25 $\quad\quad$ **for** $ty = 1$ **to** $\#tiles\_y$ **do**
26 $\quad\quad\quad t = ty + (tf - 1) \times \#tiles\_y$;
27 $\quad\quad\quad$ **if** $ty = 1$ and $tf = 1$ **then**
28 $\quad\quad\quad\quad T[t] = \max(C, I)$
29 $\quad\quad\quad$ **else if** $t = \#tiles$ **then**
30 $\quad\quad\quad\quad T[t] = \max(C, O)$
31 $\quad\quad\quad$ **else if** $ty = \#tiles\_y$ **then**
32 $\quad\quad\quad\quad T[t] = \max(C, I + W + O)$
33 $\quad\quad\quad$ **else**
34 $\quad\quad\quad\quad T[t] = \max(C, I + O)$
35 $\quad\quad\quad$ **end**
36 $\quad\quad$ **end**
37 $\quad$ **end**
38 **end**
39 $ms\_1CV = \sum_{t=1}^{\#tiles} T[t] + ms\_Mem$

[e.g., $t = 1$ in Fig. 4(a)] is only determined by the computation delay as in Algorithm 1 (line 2).

If the convolution layer has multiple tiles and all its weights are fully buffered, i.e., $Tiy < Niy$ and $Tof = Nof$, then the weights only need to be read from DRAM once and can be reused by different tiles as illustrated in Fig. 4(b). The procedure to estimate the delay of this convolution layer is summarized in Algorithm 1 (lines 3–12). The computation of the first tile [e.g., $t = 1$ in Fig. 4(b)] is overlapped with fetching the input pixels of the next tile, and there is no DMA transfer of output pixels of the previous layer, thus the delay of this tile is determined by Algorithm 1 (line 6). The computation of the last tile [e.g., $t = 3$ in Fig. 4(b)] is overlapped with transferring the output pixels of its previous tile, and its delay is calculated by Algorithm 1 (line 8). For the other tiles [e.g., $t = 2$ in Fig. 4(b)], the communication with DRAM includes both reading input pixels and writing output pixels, and the delay of one tile is expressed by Algorithm 1 (line 10). The overall delay of this convolution layer is the sum of all the tiles as well as the DRAM access delay before the first tile and after the last tile, i.e., ms_Mem.

If the convolution layer has multiple tiles and all its pixels are fully buffered, i.e., $Tiy = Niy$ and $Tof < Nof$, then the pixels only need to be read from DRAM once and can be reused by different tiles as illustrated in Fig. 4(c). Similarly, the procedure to estimate the delay of this convolution layer is summarized in Algorithm 1 (lines 13–22).

If neither the weights nor the pixels of the convolution layer can be fully buffered, i.e., $Tiy < Niy$ and $Tof < Nof$, its pipeline schedule is shown in Fig. 4(d) and the associated delay is estimated in Algorithm 1 (lines 23–37). In this case, either the pixels or the weights need to be refetched multiple times from the DRAM. In our current design, the input pixels are refetched and the weights only need to be read once. If the DRAM access requirement of input pixels is more than weights, we can also refetch weights instead and only read input pixels once by changing the DMA instructions and associated control logic. Before the computation, the first tile of weights is loaded and reused by the following consecutive $\#tiles\_y = \lceil Niy/Tiy \rceil$ tiles of pixels to perform convolution. Then, the next tile of weights is loaded and reused by the following $\#tiles\_y$ tiles of pixels. This process iterates by $\#tiles\_f = \lceil Nof/Tof \rceil$ times to complete the computation with all the $\#tiles\_f$ tiles of weights. By this means, the pixels are refetched by $\#tiles\_f$ times. A normal tile needs to read input pixels of the next tile from DRAM and write output pixels of the previous tile into DRAM, where the required weights are already loaded during the previous tile and reused. Therefore, the delay of a normal tile is estimated as in Algorithm 1

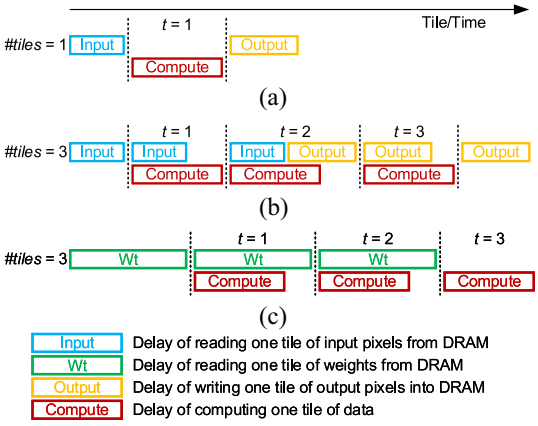


Fig. 5. Tile-by-tile delay of one pooling/FC layer, and the DRAM access delay is overlapped with the computation delay due to the dual buffering technique. (a) Max pooling: inputs fully buffered in one tile. (b) Max pooling: inputs partially buffered in multiple tiles. (c) FC.

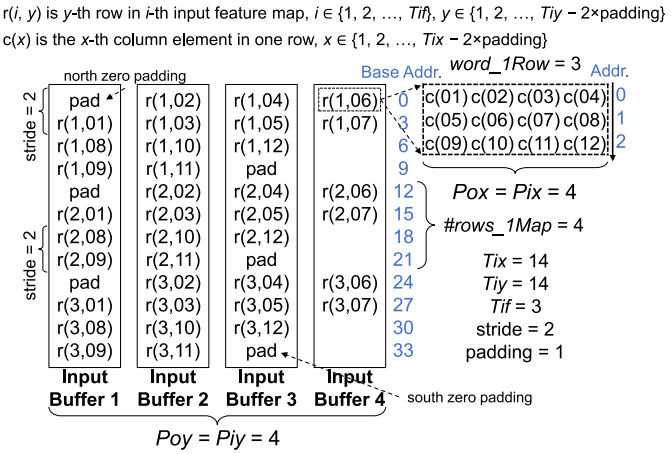


Fig. 6. Convolution data storage pattern in the input pixel buffers.

(line 34). As the first tile does not have a previous tile, there is no transfer of output pixels back to DRAM as in Algorithm 1 (line 28). For the last tile, there is no need to read input pixels for the next tile as in Algorithm 1 (line 30). When #*tiles_y* tiles of weights are finished [e.g., $ty = 3$ and $tf = 1$ in Fig. 4(d)], new tile of weights is loaded from DRAM, and DRAM access also includes transfer of pixels as in Algorithm 1 (line 32).

C. Delay Estimation of Other Layers

With the dual buffering technique employed, the overall tile-by-tile process of one max-pooling layer is illustrated in Fig. 5(a) and (b), which is similar to the convolution layer except that pooling does not need weights. If the pooling layer has only one tile, which means the inputs of one pooling layer can be fully buffered, there is no overlapping between memory transfer and computation as shown in Fig. 5(a) and (b) illustrates the dual buffering pipeline of one pooling layer with multiple tiles. Similar to Algorithm 1, we can compute the overall latency of max-pooling layers according to the tile-by-tile execution schedule, with the delay of max-pooling computation and DRAM access calculated similar to the convolution layer.

Fig. 5(c) shows the pipeline schedule of FC layer, where weights are fetched before the corresponding computation and no outputs are transferred back to DRAM. The storage format of FC weights in the weight buffer allows us to read *Pof* weights simultaneously every clock cycle to parallel compute *Pof* outputs. Then, the computation cycles of one FC tile equal to the depth of buffered FC weights. The overall delay of FC is bounded and determined by the computation delay or the DRAM access delay of weights.

VI. Size Requirement of On-Chip Memory

With the specific data storage pattern of buffers, we can more precisely calculate the required on-chip buffer sizes than the rough estimation in Section III-B.

A. Size and Storage of Input Buffers

Fig. 6 illustrates the proposed storage pattern of convolution input pixels, which benefits the dataflow of $Pox \times Poy$ pixels from buffers into MAC units [11]. The width of one input buffer is determined by *Pox* to feed data for parallel computation of *Pox* pixels in one feature map row. The number of input buffers is determined by *Poy* to feed data for parallel computation of *Poy* multiple output rows. In Fig. 6, $c(x)$ denotes one input pixel in the xth column of a certain row, where $x \in \{1, 2, \ldots, Tix - 2 \times \text{padding}\}$ and *Tix* includes both the east and west zero padding. The east and west zero paddings are not stored in buffers and instead they are masked out by control logic before loading into the MAC units. The number of addresses or words occupied by one row is

$$word\_1Row = \lceil (Tix - 2 \times \text{padding})/Pox \rceil. \tag{18}$$

In Fig. 6, $r(i, y)$ is the yth row of the ith input feature map, where $i \in \{1, 2, \ldots, Tif\}$ and $y \in \{1, 2, \ldots, Tiy\}$. The *Tiy* rows of one input feature map, including north and south zero paddings if they exist, are distributed across the *Poy* number of input buffers. With stride $= 2$ as in Fig. 6, two adjacent rows are continuously stored in the same buffer according to the dataflow requirement. Then, the number of rows of one feature map, i.e., $r(i, y)$, in one buffer is

$$\#rows\_1Map = \lceil \lceil Tiy/\text{stride} \rceil / Poy \rceil \times \text{stride}. \tag{19}$$

The storage location of the subsequent input feature maps is aligned with the first feature map to simplify the address generation logic, which causes some overhead due to the non-continuous storage pattern as shown by the blank spaces in the buffers in Fig. 6. By this means, the depth or words requirement of one input buffer (*InBuf*) storing *Tif* input feature maps for one convolution layer is expressed as

$$word\_InBuf = word\_1Row \cdot \#rows\_1Map \cdot Tif. \tag{20}$$

The data width of one input buffer is $Pox \times bit\_Px$ and the number of input buffers is $Poy \times$ Dual with Dual $= 2$, where Dual represents doubling of the number of buffers due to the dual buffer structure. Therefore, in every clock cycle, $Pox \times Poy$

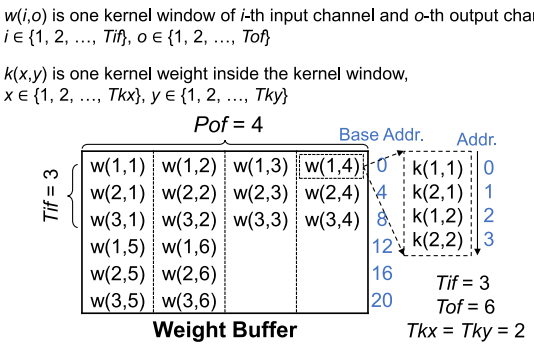


Fig. 7. Convolution data storage pattern in the weight buffer.

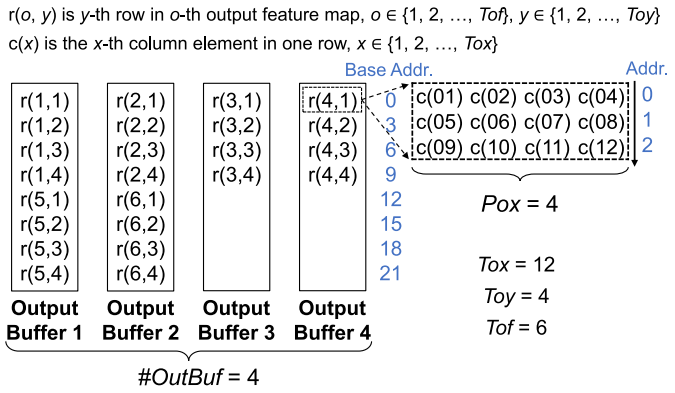


Fig. 8. Convolution data storage pattern in the output pixel buffers.

pixels can be fed into the MAC units. The input buffer size requirement of one convolution layer is

$$bit\_InBuf = \text{Dual} \times Poy \times Pox \times bit\_Px \times word\_InBuf. \tag{21}$$

The final input buffer size is the maximum bit_InBuf of all the convolution layers. The actual input buffer size in (21) is larger than the rough estimation in (5) due to the mismatch of tile and buffer dimensions caused by the specific storage pattern.

B. Size and Storage of Weight Buffers

The storage pattern of weight buffer is illustrated in Fig. 7. The $k(x, y)$ in Fig. 7 denotes one weight inside the $Nkx \times Nky$ kernel window, where $x \in \{1, 2, \ldots, Tkx\}$ and $y \in \{1, 2, \ldots, Tky\}$. In the chosen design, we always have $Tkx = Nkx$ and $Tky = Nky$, so that one kernel window is fully buffered. These $Tkx \times Tky$ weights, i.e., $k(x, y)$, are stored in continuous addresses as we serially compute one kernel window, e.g., $Pkx = Pky = 1$. In Fig. 7, $w(i, o)$ denotes one kernel window of the ith input channel and oth output channel, which is comprised of $Tkx \times Tky$ weights. Weights from different input channels (Tif) are stacked in different addresses as we serially compute each input channel. To compute Pof output channels in parallel, the weights of Pof output channels are stored at the same address of the weight buffer. Therefore, the bit width of the weight buffer is $Pof \times bit\_Wt$. The words or depth of the weight buffer ($WtBuf$) is

$$word\_WtBuf = Tkx \times Tky \times Tif \times \lceil Tof/Pof \rceil. \tag{22}$$

With dual buffering, the number of weight buffers is two. The weight buffer size requirement of one convolution layer is

$$bit\_WtBuf = \text{Dual} \cdot Pof \cdot bit\_Wt \cdot word\_WtBuf. \tag{23}$$

If Tof/Pof is not an integer, some blank spaces in the weight buffer are wasted as in Fig. 7. The final weight buffer size is the maximum bit_WtBuf of all the convolution layers.

C. Size and Storage of Output Buffers

After every $Nkx \times Nky \times Nif$ clock cycles, there are $Pox \times Poy \times Pof$ outputs from MAC units. To reduce the bit width of data bus and the bandwidth requirement of output buffers as in Fig. 8, the parallel outputs are serialized into $Poy \times \lceil Pof/\#OutBuf \rceil$ clock cycles, where $\#OutBuf$ is the number of output buffers excluding the dual buffer structure with $\#OutBuf \leq Pof$. By this means, the data width of one output buffer is $Pox \times bit\_Px$, as shown in Fig. 8, to store the parallel Pox outputs from the same feature map.

The output buffer storage pattern is illustrated in Fig. 8, where $c(x)$ is the xth column element in one row with $x \in \{1, 2, \ldots, Tox\}$ and $r(o, y)$ is the yth row in the oth output feature map with $o \in \{1, 2, \ldots, Tof\}$ and $y \in \{1, 2, \ldots, Toy\}$. The outputs of the same feature map are continuously stored in the same buffer in a row-major order. One row ($r(o, y)$) is comprised of Tox elements ($c(x)$) continuously stored in $\lceil Tox/Pox \rceil$ addresses, and we set $Tox = Nox$ so that one entire row is processed while maintaining the row-major order. One feature map has Toy number of rows stored in one buffer and it occupies $Toy \times \lceil Tox/Pox \rceil$ addresses. One output buffer stores $\lceil Tof/\#OutBuf \rceil$ number of feature maps. Then, the number of words or the depth of one output buffer ($OutBuf$) for one convolution layer is

$$word\_OutBuf = \lceil Tof/\#OutBuf \rceil Toy \lceil Tox/Pox \rceil. \tag{24}$$

The output buffer size requirement of one convolution layer is

$$bit\_OutBuf = (\text{Dual} \times \#OutBuf) \times (Pox \times bit\_Px) \times word\_OutBuf. \tag{25}$$

If $Tof/\#OutBuf$ is not an integer, the blank spaces in the output buffers as in Fig. 8 are wasted.

D. Size and Storage of Pooling Buffers

The max-pooling layers share the input and output buffers with convolution layers. Due to the different dataflow requirement, the max-pooling input storage pattern in the input buffers is different from convolution inputs, but it is the same as the output storage pattern of convolution outputs in Fig. 8. In addtion, the output buffer storage pattern of max-pooling layers is also the same as the convolution outputs in Fig. 8. The pixels from the same feature map are stored in the same buffer, and different feature maps are distributed across different buffers. Therefore, the input and output buffer depth of one tile of max pooling is similar to (24). The buffer size requirement of pooling layers is ensured to be smaller than

that of the convolution layers by using smaller pooling tiling variables so that there is no overflow of pooling data.

VII. Modeling of On-Chip Buffer Access

The energy cost of accessing data in the buffers dominates the on-chip memory energy consumption [18], [19], so it is essential to reduce the size of buffer accesses for energy-efficient design. To reduce the buffer access size, data should be reused as much as possible either by multiple PEs or by different execution tiles, which will be discussed in this section.

A. Reading Input and Weight Buffers of Convolution

Based on (9) to estimate the buffer access, we need to compute #*cycles_Access* first. In this case, #*cycles_Access* is the MAC computation clock cycles of one tile, which is #*cycles*_1*T* in (15). Then, the computation clock cycles of all the convolution layers are

$$\#cycles\_C = \sum_{L=1}^{\#\text{CONVs}} \#cycles\_1T[L] \times \#tiles[L] \tag{26}$$

where #CONVs is the number of convolution layers and #*tiles* is the number of tiles. The size (bit) of data read (*Rd*) from input buffers (*InBuf*) for convolution layers is computed by multiplying the read clock cycles with the total input buffer data width as

$$bit\_RdInBuf = \#cycles\_C \cdot (Pox \cdot Poy \cdot bit\_Px) \tag{27}$$

where every $Pox \times Poy$ pixels are reused by *Pof* MAC units and the number of input buffer accesses is reduced by *Pof* times. Similarly, the size (bit) of data read (*Rd*) from weight buffers (*WtBuf*) for all the convolution layers is

$$bit\_RdWtBuf = \#cycles\_C \times (Pof \times bit\_Wt) \tag{28}$$

where every *Pof* weights are reused by $Pox \times Poy$ MAC units and the number of weight buffer accesses is reduced by $Pox \times Poy$ times.

B. Writing Input and Weight Buffers of Convolution

Before computation, the input data are written into the input and weight buffers from DMA. As discussed in Section V-B, not every tile needs to read both pixels and weights from DRAM, because some pixels or weights of one tile can be reused by the following adjacent tiles. The number of tiles of one convolution layer that write new weights (*Wt*) to the weight buffer is

$$\#tiles\_Wt = \lceil Nof/Tof \rceil. \tag{29}$$

The number of tiles of one convolution layer that write new input pixels (*In*) to the input buffers is

$$\#tiles\_In = \begin{cases} \lceil \frac{Noy}{Toy} \rceil \lceil \frac{Nof}{Tof} \rceil, & \text{if } Toy < Noy \text{ and } Tof < Nof \\ \lceil \frac{Noy}{Toy} \rceil, & \text{otherwise.} \end{cases} \tag{30}$$

When neither weights nor pixels are fully buffered, i.e., $Toy < Noy$ and $Tof < Nof$, the same pixels are reloaded $\lceil Nof/Tof \rceil$ times into input buffers as shown in Fig. 4(d). Similar to (21), the size (bit) of one tile (1*T*) of pixels written into the input buffers is

$$bit\_WrIn\_1T = word\_InBuf \cdot Poy \cdot Pox \cdot bit\_Px. \tag{31}$$

The size (bit) of data loaded into the input buffers of all the convolution layers is

$$bit\_WrInBuf = \sum_{L=1}^{\#\text{CONVs}} bit\_WrIn\_1T[L] \times \#tiles\_In[L]. \tag{32}$$

Similarly, the size (bit) of one tile of weights written into the weight buffers is

$$bit\_WrWt\_1T = word\_WtBuf \times Pof \times bit\_Wt \tag{33}$$

and the size (bit) of data written into the weight buffers of all the convolution layers is

$$bit\_WrWtBuf = \sum_{L=1}^{\#\text{CONVs}} bit\_WrWt\_1T[L] \times \#tiles\_Wt[L]. \tag{34}$$

C. Data Access of Output Buffers of Convolution

The number of clock cycles to write outputs into output buffers during one tile is the same as *word_OutBuf*, where one word of data is written into one output buffer in one cycle. Since every tile of one layer has outputs to be saved, the clock cycles of writing outputs to output buffers are *word_OutBuf* × #*tiles*. Then, the total cycles to load outputs into output buffers (*OutBuf*) are summed up across all the convolution layers as

$$\#cycles\_WrOutBuf = \sum_{L=1}^{\#\text{CONVs}} word\_OutBuf[L] \times \#tiles[L]. \tag{35}$$

The size (bit) of results written into the output buffers is

$$bit\_WrOutBuf = \#cycles\_WrOutBuf \times \#OutBuf \times Pox \times bit\_Px. \tag{36}$$

Since each output is written into and read from the output buffers only once, the size (bit) of data read from output buffers (*bit_RdOutBuf*) by DMA equals to *bit_WrOutBuf*.

VIII. Experiments and Analysis

In this section, the proposed performance model is used to explore the design space by tuning the key design variables, e.g., unrolling and tiling sizes, and DRAM bandwidth and accelerator frequency, to identify the performance bottleneck and obtain the optimal design configurations.

A. Design Space Exploration of Tilling Variables

The loop tiling strategy determines how much data of each layer is buffered, which affects the buffer capacity requirement, the number of DRAM accesses, and the accelerator performance. Although we have fixed $Tkx = Nkx$, $Tky = Nky$, $Tif = Nif$, and $Tox = Nox$ as mentioned in Section II-D, the

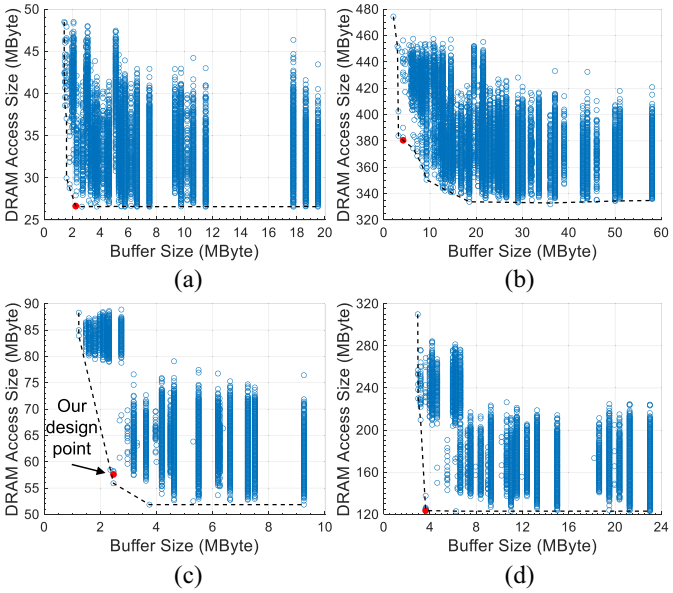


Fig. 9. Tiling variables (*Toy* and *Tof*) are swept to explore the relationship between the size of DRAM accesses and the total input/weight/output buffer size requirement, where $Pox \times Poy \times Pof = 7 \times 7 \times 32$ with 16-bit data. (a) NiN. (b) VGG-16. (c) GoogLeNet. (d) ResNet-50.

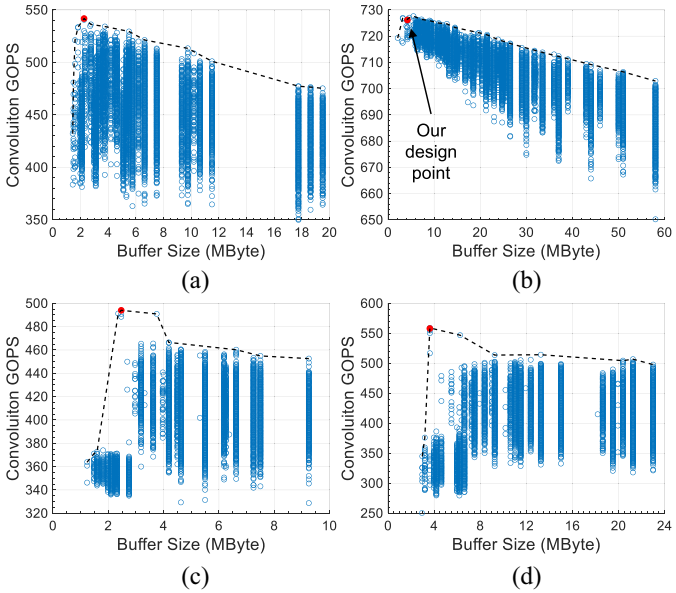


Fig. 10. Tiling variables (*Toy* and *Tof*) are swept to explore the relationship between the convolution throughputs and the total input/weight/output buffer size requirement, where $Pox \times Poy \times Pof = 7 \times 7 \times 32$, $MHz\_Accelerator = 240$, $BW\_DRAM = 14.4$ GB/s. (a) NiN. (b) VGG-16. (c) GoogLeNet. (d) ResNet-50.

remaining two tiling variables *Toy* and *Tof* still give us a huge design space as mentioned in [11]. For example, VGG-16 has 13 convolution layers, and there are $13 \times 2 = 26$ tiling variables and each variable can have 4 or more candidate values determined by Noy/Poy or Nof/Pof, then the total number of *Toy* and *Tof* choices is roughly $4^{26} = 4.5 \times 10^{15}$, which results in an enormous solution space that cannot be enumerated. Therefore, we randomly sample 30 000 tiling configurations for different CNN algorithms to explore their impact on the memory access and performance as in Figs. 9–11, where we set loop unrolling variables as $Pox \times Poy \times Pof = 7 \times 7 \times 32$.

The relationship between tiling variables and the number of DRAM accesses is investigated in Fig. 9 with 16-bit data. The total convolution DRAM access size is computed by

$$byte\_DRAM = \sum_{L=1}^{\#CONVs} (byte\_RdPx \cdot \#tiles\_In + byte\_RdWt \times \#tiles\_Wt + byte\_WrPx \cdot \#tiles) \quad (37)$$

where the right-hand side variables are computed by (11), (12), (14), (29), and (30). The DRAM accesses of other layers are also included in Fig. 9. One circle in Fig. 9 represents one design point of the tiling variables *Toy* and *Tof*. Since the buffer size is determined by the layer with the maximum tiling size, there could be multiple different tiling configurations in other layers leading to the same buffer size. The buffer size in Fig. 9 includes input/weight/output buffers, which equals to $\max(bit\_InBuf) + \max(bit\_WtBuf) + \max(bit\_OutBuf)$ from (21), (23), and (25). With the increase of tiling and buffer sizes, the number of DRAM accesses is decreasing as shown by the dashed line in Fig. 9. After the buffer size is increased to be large enough, we can achieve the minimum DRAM accesses. The red dot in Fig. 9 is our optimal design choice of *Toy* and *Tof* that balances the buffer size requirement and the number of DRAM accesses.

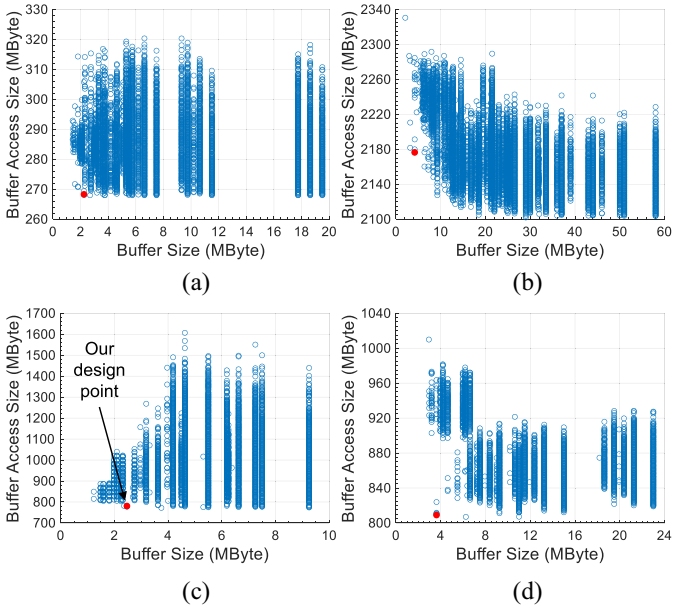


Fig. 11. Tiling variables (*Toy* and *Tof*) are swept to explore the relationship between the size of on-chip buffer accesses and the size requirement of buffers, where $Pox \times Poy \times Pof = 7 \times 7 \times 32$. (a) NiN. (b) VGG-16. (c) GoogLeNet. (d) ResNet-50.

Fig. 10 shows the relationship between tiling sizes and the convolution throughputs, where the accelerator operating frequency is 240 MHz and the DRAM bandwidth is 14.4 GB/s. The throughput is computed by $\#operations/delay$, where $\#operations = 2$ Nm including both multiply and addition, and *delay* is the sum of ms_1CV over all the convolution

layers. If the tiling or buffer size is too small, the number of DRAM access and the associated latency is significantly increased, which degrades the throughput. If the tiling size is too large or there is only one tile in one layer, the DRAM access latency cannot be well overlapped with the computation delay as mentioned in Section V-B, which results in lower throughput. This trend is shown by the dashed line in Fig. 10. The dashed lines of GoogLeNet and ResNet-50 are not as smooth as those of NiN and VGG-16. It is mainly because GoogLeNet and ResNet-50 have more layers resulting in much larger design space, which makes it more difficult to cover all the design choices through random sampling. The red dots in Fig. 10 are our design choices of *Toy* and *Tof*, which are the same in Fig. 9, to achieve the best throughputs.

Fig. 11 shows the relationship between tiling sizes and the number of on-chip buffer accesses for different CNN algorithms, which include both read and write operations of input/weight/output buffers of all the layers in a given CNN algorithm. Based on our acceleration strategy [11], the partial sums are accumulated inside the MAC units, which do not involve buffer access. The estimation of the number of on-chip buffer accesses is discussed in Section VII. Our design choices of *Toy* and *Tof* shown by red dots in Fig. 11 can achieve close to the optimal number of buffer accesses while having best throughputs and low level of DRAM accesses.

B. Design Space Exploration for Performance

As convolution dominates the CNN operations [2]–[4], [20], we focus on the design space exploration of convolution throughputs. The convolution throughput is affected by several factors, namely the accelerator operating frequency, external memory bandwidth, and the loop unrolling variables, these are explored in Fig. 12 using GoogLeNet as an example. With a small number of MAC units and high DRAM bandwidth (*BW_DRAM*) as shown in Fig. 12(a), the accelerator throughput is mainly bounded by computation, and thus the throughput is almost linearly increasing with the frequency when $BW\_DRAM > 12.8$ GB/s. If the DRAM bandwidth is too low, e.g., 3.2 GB/s, the design is more likely to be memory bounded and the throughput stops increasing with the frequency. With more MAC units and higher frequency, the throughputs are tend to increase, as shown in Fig. 12, until the design touches the memory roof which is illustrated in Fig. 13.

The memory roof throughput [7] in Fig. 13 is the maximum achievable throughput under a certain external memory bandwidth and it is defined as

$$DRAM\_roof(GOPS) = \frac{\#operations(GOP)}{DRAM\_delay(s)} = \frac{\#operations(GOP)}{\#data(GByte)} BW\_Memory(GB/s) \tag{38}$$

where *#data* is the data size of DRAM accesses. Since the computation-to-communication (CTC) ratio, i.e., *#operations*/*#data*, is a constant under a certain tiling setting, *DRAM_roof* is directly proportional to *BW_Memory*. With the same setting of *BW_Memory* for GoogLeNet and VGG-16, the shape of the curves in Fig. 13(a) and (b) is similar. Since VGG-16 has a higher CTC, its memory roof throughput is much higher than GoogLeNet in Fig. 13. As discussed in Section IV-B, the memory bandwidth (*BW_Memory*) is bounded by both the DRAM controller (*BW_DRAM*) and DMA (*BW_DMA*). At low frequency, *BW_Memory* is limited by *BW_DMA*, and *DRAM_roof* is linearly increasing with the increase of frequency as in Fig. 13. After *BW_DMA* is larger than *BW_DRAM*, *BW_Memory* is limited by *BW_DRAM* instead, and *DRAM_roof* stops growing with the frequency. The saturated throughputs in Fig. 12 are lower than *DRAM_roof* in Fig. 13, which is mainly because there are redundant DRAM transfers and the computation delay is not fully overlapped with DRAM latency.

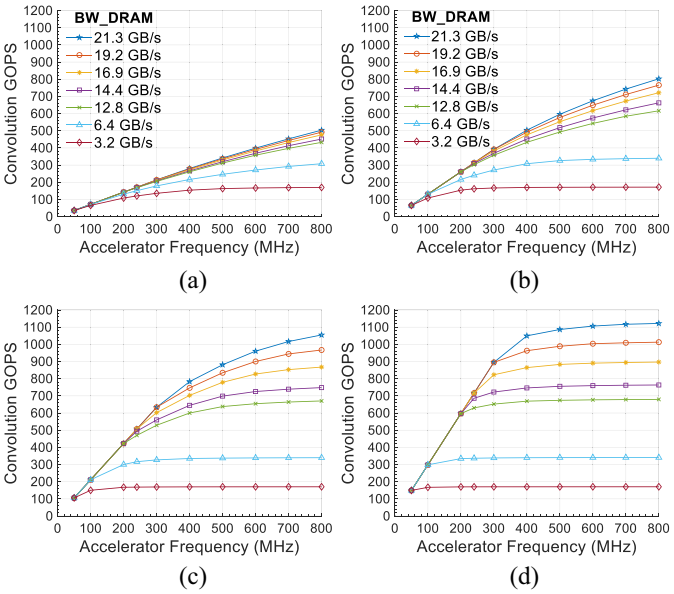


Fig. 12. Convolution throughput is affected by the accelerator operating frequency, DRAM bandwidth, and the number of MAC units. GoogLeNet is shown as an example here. (a) $Pox = 7, Poy = 7$, and $Pof = 8$. (b) $Pox = 7, Poy = 7$, and $Pof = 16$. (c) $Pox = 7, Poy = 7$, and $Pof = 32$. (d) $Pox = 14, Poy = 7$, and $Pof = 32$.

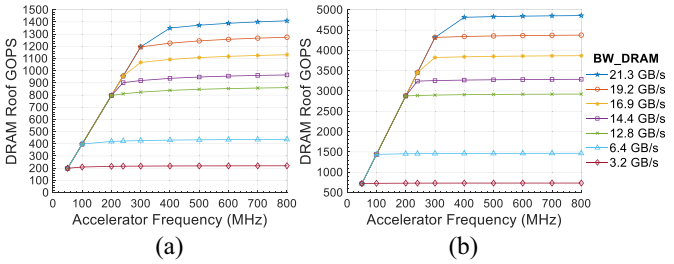


Fig. 13. External memory roof throughput (*DRAM_roof*) is the maximum achievable throughput under a certain memory bandwidth. (a) GoogLeNet. (b) VGG-16.

C. Performance Model Validation

Fig. 14 shows the comparison of throughput and latency between the performance model and the on-board test results on Arria 10 and Stratix 10 with different number of MAC units, where both pixels and weights are 16-bit fixed point

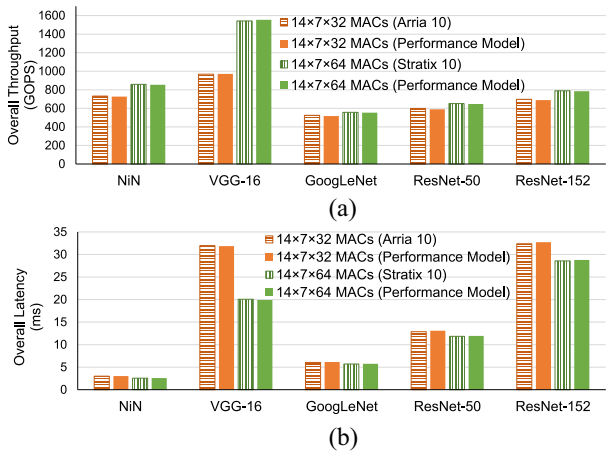


Fig. 14. Performance model results are compared with on-board test results of Arria 10 and Stratix 10 on overall (a) throughput and (b) latency.

data. The differences between the estimation and on-board results are within 3%, which are mainly due to the DRAM transfer latency mismatch, minor layers (e.g., average pooling), and some pipeline stages in the real implementation. The compilation of our FPGA design using Quartus Pro 17.1 on 16-core Intel Xeon CPU E5-2650 v3 normally takes 6–8 h, while the performance model running on Laptop Intel Core i7-7500U CPU using MATLAB takes about 1–5 s per design.

D. Related Works

Several related works have used performance model to optimize the memory access and computation pattern of their proposed architecture and dataflow. Suda *et al.* [8] implemented convolution as matrix multiplication and uses a performance model to optimize the design. However, the execution time in [8] only counts computation time without considering the DRAM transfer latency. If the design becomes memory bounded, the model in [8] cannot properly predict the overall latency, which results in the estimation discrepancy of FC layers with high computation parallelism. The proposed systolic array architecture in [10] is also optimized through a performance model. The overall throughput is simply computed by the minimum of the computation throughput and DRAM transfer throughput, where the overlap efficiency of computation and data transfer is not considered. The fine-grained tile-level data accesses of DRAM and buffers are not explored in [10]. The buffer and DRAM accesses are modeled in [18] to explore different data reuse patterns by changing the tiling strategy and computation order. Only coarse-grained modeling of the convolution memory access is analyzed without considering the DRAM bandwidth utilization and the detailed data storage patterns in buffers and DRAM. The proposed hybrid data reuse in [18] is similar to our tiling strategy that different layers can use different tiling sizes to either reuse weights or pixels to minimize the DRAM access. In this paper, the relationship between the overall DRAM access and the total buffer size is also investigated. The power of data movement in different hierarchy, e.g., DRAM, buffer, and PE array, is analytically modeled in [19] to compare the energy efficiency of different dataflow. However, the power is not quantitatively formulated with the design variables in [19], and the performance of the accelerator is not modeled.

IX. Further Improvement Opportunities

In this section, we use the proposed performance model to evaluate the opportunities that may further enhance the performance of the accelerator by improving the efficiency of DRAM transactions and DSP utilization.

A. Improving DRAM Bandwidth Utilization

To simplify the control logic of data bus from DMA to input buffers, different feature map rows are aligned in different addresses in our current design. By this means, if the number of pixels in one row is smaller than $\lfloor bit\_DMA/bit\_Px \rfloor$, the successive row directly starts from the next address instead of continuously using the same address resulting in the waste of DMA datawidth. For example, with $bit\_Px = 16$, one address can accommodate $512/16 = 32$ pixels, if the width of the feature map is $Nix = 14$, then the actual number of pixels of one row read from DRAM in (11) is $Tix = 32$, where $32 - 14 = 20$ data are redundant. Some CNN models, e.g., GoogLeNet and ResNet, have a lot of convolution layers with small *Nix*, e.g., 7 or 14, then their throughputs are significantly affected by the inefficient utilization of DMA datawidth.

To improve the DRAM bandwidth utilization, one method is to store multiple rows in one DMA address, which involves the modifications of control logic and extra data paths from DMA to input buffers. The other method is to keep the data aligned, but narrow the bit width of the data bus between DMA and input buffers. To attain the same data transfer rate, higher frequency is needed, and asynchronous FIFO may be used. In the performance model, we reduce *bit_DMA* to be 256 and 128 and increase their corresponding frequency of the data bus to predict the throughput improvements. In Fig. 15, our current design (DMA 512-bit) serves as the baseline with data aligned, and *bit_DMA* is set to be 256 or 128, which has the same effect as supporting two or four rows in one address with $bit\_DMA = 512$, respectively. Fig. 15 shows that NiN, GoogLeNet, and ResNet can benefit a lot from decreasing the DMA bit width, mainly because they have many layers with small *Nix* and the layers with small *Nix* are memory bounded. On the contrary, VGG-16 cannot benefit from higher DRAM bandwidth utilization as the design is still computationally bounded. Based on the prediction, it is compelling to improve our design for higher DRAM bandwidth utilization.

B. Merging the First Layers

In GoogLeNet and ResNet, there are multiple parallel branches of layers, and the first layer of each branch reads input pixels from the same precedent layer. If these convolution layers also have the same kernel size and stride, they can be merged into one layer along the output feature map dimension (*Nof*). By this means, the input pixels can be shared by the first layers of different branches and only need to be read

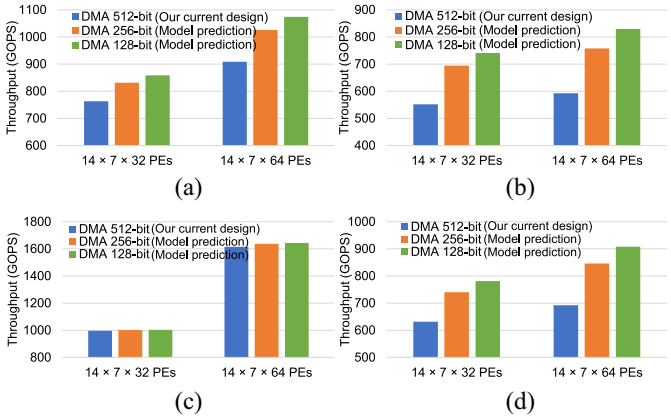


Fig. 15. Performance model predicts that the throughput will be improved by increasing the DRAM bandwidth utilization, which is achieved by decreasing the DMA bit width to reduce the redundant DRAM accesses. (a) NiN. (b) GoogLeNet. (c) VGG-16. (d) ResNet-50.

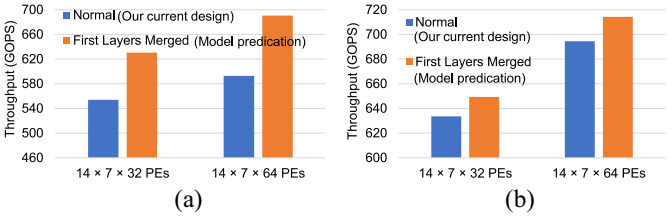


Fig. 16. Performance model predicts that the throughput will be improved by merging the first layers of different parallel branches, which read from the same precedent layer, to eliminate the repeated DRAM access, where "Normal" denotes our current design as baseline. (a) GoogLeNet. (b) ResNet-50.

from DRAM once, as proposed in [13]. We change the corresponding settings of our performance model, e.g., *byte_RdPx* in (11), to estimate the effect of eliminating the repeated DRAM accesses of the precedent layer as shown in Fig. 16. Since GoogLeNet and ResNet are already memory bounded in our current design, reducing the DRAM access can considerably improve the throughputs. The required modifications of our current design to merge the first layers involve changing the control logic and the descriptors of DMA transactions, and there is no significant overhead of additional hardware resources.

C. Improving PE Efficiency

Due to the highly varying dimensions of different convolution layers in a given CNN model, it is a challenge task to efficiently distribute workloads across PEs, or we need to make loop dimensions (N$*$) divisible by their corresponding unrolling variables ($P*$). In [21] and [22], an adaptive parallelism scheme is proposed to dynamically adjust the mapping of operations on different PEs, or the unrolling variables can be changed for each layer to maximize PE utilization. This requires the ability to dynamically redirect data flow from buffers to PEs, which may need complex control logic, incur penalty of additional resources, and aggravate the burden on timing closure.

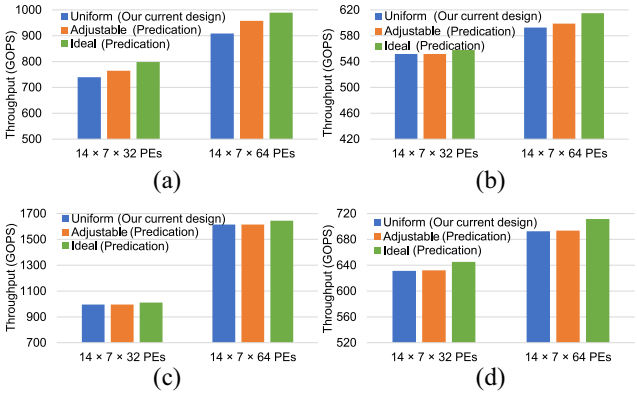


Fig. 17. Uniform: our current design as baseline with uniform PE mapping; Adjustable: dynamically adjust the unrolling variables for different layers to improve PE utilization; Ideal: force PE utilization to be 100%. (a) NiN. (b) GoogLeNet. (c) VGG-16. (d) ResNet-50.

Instead of using uniform PE mapping and unrolling variables in the current design, we adjust unrolling variables ($Pox \cdot Poy \cdot Pof$) for different layers to achieve better PE utilization in the performance model as shown by "Adjustable" in Fig. 17. We also force PE utilization to be 100% by removing the ceiling functions in (2), which is denoted by "Ideal" in Fig. 17. However, the throughput improvements from adjustable unrolling strategy are very limited (<10%) for our design, mainly because: 1) the $Nox \cdot Noy \cdot Nof$ dimensions of most layers have already been able to provide large enough parallelism for our uniform unrolling strategy and 2) most of our layers are memory bounded and the reduction of computation latency has little effect on the throughput. Considering the large amount of necessary design efforts for adjustable PE mapping and low expected improvements, we surmise it is not a primary task in our future work to adopt this technique.

X. Conclusion

In this paper, a high-level performance model is proposed to estimate the key specifications, e.g., throughput, of FPGA accelerators for CNN inference, which enables the design space exploration to identify performance bottleneck in the early development phase. The design strategy and resource costs are formulated using the design variables of loop unrolling and tiling. The proposed performance model is validated for a specific acceleration strategy across a variety of CNN algorithms comparing with on-board test results on two different FPGAs.

References

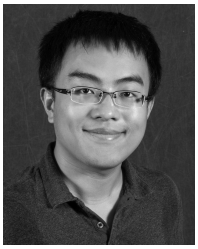
[1] O. Russakovsky *et al.*, "ImageNet large scale visual recognition challenge," *Int. J. Comput. Vis.*, vol. 115, no. 3, pp. 211–252, 2015.

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

[3] K. Simonyan and A. Zisserman. (2014). *Very Deep Convolutional Networks for Large-Scale Image Recognition*. [Online]. Available: http://arxiv.org/abs/1409.1556

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

[5] C. Szegedy *et al.*, "Going deeper with convolutions," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Boston, MA, USA, Jun. 2015, pp. 1–9.
[6] C. Szegedy, S. Ioffe, V. Vanhoucke, and A. A. Alemi, "Inception-v4, inception-ResNet and the impact of residual connections on learning," in *Proc. Conf. Artif. Intell. (AAAI)*, Feb. 2017, pp. 4278–4284.
[7] C. Zhang *et al.*, "Optimizing FPGA-based accelerator design for deep convolutional neural networks," in *Proc. ACM/SIGDA Int. Symp. Field Program. Gate Arrays (FPGA)*, Monterey, CA, USA, 2015, pp. 161–170.
[8] 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)*, Monterey, CA, USA, 2016, pp. 16–25.
[9] C. Zhang, Z. Fang, P. Zhou, P. Pan, and J. Cong, "Caffeine: Towards uniformed representation and acceleration for deep convolutional neural networks," in *Proc. Int. Conf. Comput.-Aided Design (ICCAD)*, Austin, TX, USA, Nov. 2016, pp. 1–8.
[10] X. Wei *et al.*, "Automated systolic array architecture synthesis for high throughput CNN inference on FPGAs," in *Proc. Design Autom. Conf. (DAC)*, Austin, TX, USA, 2017, pp. 1–6.
[11] Y. Ma, Y. Cao, S. Vrudhula, and J.-S. Seo, "Optimizing the convolution operation to accelerate deep neural networks on FPGA," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 26, no. 7, pp. 1354–1367, Jul. 2018.
[12] H. Zeng, R. Chen, C. Zhang, and V. K. Prasanna, "A framework for generating high throughput CNN implementations on FPGAs," in *Proc. ACM/SIGDA Int. Symp. Field Program. Gate Arrays (FPGA)*, Monterey, CA, USA, Feb. 2018, pp. 117–126.
[13] X. Lin *et al.*, "LCP: A layer clusters paralleling mapping method for accelerating inception and residual networks on FPGA," in *Proc. Design Autom. Conf. (DAC)*, San Francisco, CA, USA, Jun. 2018, pp. 1–6.
[14] K. Pavel and S. David, "Algorithms for efficient computation of convolution," in *Design and Architectures for Digital Signal Processing*. New York, NY, USA: IntechOpen, Jan. 2013. doi: 10.5772/51942.
[15] J. Yu *et al.*, "Real-time object detection towards high power efficiency," in *Proc. Design Autom. Test Eur. Conf. Exhibit. (DATE)*, Dresden, Germany, Mar. 2018, pp. 704–708.
[16] C. Zhang and V. K. Prasanna, "Frequency domain acceleration of convolutional neural networks on CPU-FPGA shared memory system," in *Proc. ACM/SIGDA Int. Symp. Field Program. Gate Arrays (FPGA)*, Monterey, CA, USA, 2017, pp. 35–44.
[17] 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. Int. Conf. Field Program. Logic Appl. (FPL)*, Ghent, Belgium, Sep. 2017, pp. 1–8.
[18] F. Tu *et al.*, "Deep convolutional neural network architecture with reconfigurable computation patterns," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 25, no. 8, pp. 2220–2233, Aug. 2017.
[19] Y.-H. Chen, J. S. 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)*, Seoul, South Korea, Jun. 2016, pp. 367–379.
[20] M. Lin, Q. Chen, and S. Yan, "Network in network," *CoRR*, vol. abs/1312.4400, 2013. [Online]. Available: http://arxiv.org/abs/1312.4400
[21] L. Song *et al.*, "C-Brain: A deep learning accelerator that tames the diversity of CNNs through adaptive data-level parallelization," in *Proc. Design Autom. Conf. (DAC)*, Austin, TX, USA, Jun. 2016, pp. 1–6.
[22] M. Putic *et al.*, "Dyhard-DNN: Even more DNN acceleration with dynamic hardware reconfiguration," in *Proc. Design Autom. Conf. (DAC)*, San Francisco, CA, USA, Jun. 2018, pp. 1–6.



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 pursuing the Ph.D. degree with Arizona State University, Tempe, AZ, USA.

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

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

He was a Summer Intern with Hewlett-Packard Labs, Palo Alto, CA, USA, in 2000, and with IBM Microelectronics Division, East Fishkill, NY, USA, in 2001. He was a Post-Doctoral Researcher with the Berkeley Wireless Research Center, Berkeley. He is currently a Professor of electrical engineering with Arizona State University, Tempe, AZ, USA. He has published 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 post-silicon technologies, and hardware design for on-chip learning.

Dr. Cao was a recipient of the 2012 Best Paper Award at IEEE Computer Society Annual Symposium on Very Large Scale Integration, the 2010, 2012, 2013, 2015, and 2016 Top 5% Teaching Award at Schools of Engineering, Arizona State University, the 2009 ACM SIGDA Outstanding New Faculty Award, the 2009 Promotion and Tenure Faculty Exemplar at Arizona State University, the 2008 Chunhui Award for Outstanding Oversea Chinese Scholars, the 2007 Best Paper Award at the 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 has served as Associate Editor for the IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS, and served on the technical program committees of many conferences. He was a Distinguished Lecturer of the IEEE Circuits and Systems Society in 2009.

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 with the ECE Department, University of Arizona, Tucson, AZ, USA, and was on the Faculty of the EE-Systems Department, University of Southern California. He was also the Founding Director of the NSF Center for Low Power Electronics, University of Arizona, Tempe, AZ, USA, where he is a Professor of Computer Science and Engineering, and the Director of the NSF I/UCRC Center for Embedded Systems. More recently, he has been investigating non-conventional 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. His current research interests include design automation and computer aided design for digital integrated circuit and systems, focusing on low power circuit design, and energy management of circuits and systems, energy optimization of battery-powered computing systems, including smartphones, wireless sensor networks, and IoT systems that rely on 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, new circuit architectures of threshold logic circuits for the design of application-specified integrated circuits, and field-programmable gate array.

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 IBM T. J. Watson Research Center, Yorktown Heights, NY, USA, where he researched on 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 Laboratory as a Visiting Faculty Member. 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.