Efficient Hardware Architectures for Deep Convolutional Neural Network

Publisher: IEEE Cite This PDF

Jichen Wang ; Jun Lin ; Zhongfeng Wang **All Authors**

116 Cites in Papers **7634** Full Text Views

Abstract:
Convolutional neural network (CNN) is the state-of-the-art deep learning approach employed in various applications. Real-time CNN implementations in resource limited embedded systems are becoming highly desired recently. To ensure the programmable flexibility and shorten the development period, field programmable gate array is appropriate to implement the CNN models. However, the limited bandwidth and on-chip memory storage are the bottlenecks of the CNN acceleration. In this paper, we propose efficient hardware architectures to accelerate deep CNN models. The theoretical derivation of parallel fast finite impulse response algorithm (FFA) is introduced. Based on FFAs, the corresponding fast convolution units (FCUs) are developed for the computation of convolutions in the CNN models. Novel data storage and reuse schemes are proposed, where all intermediate pixels are stored on-chip and the bandwidth requirement is reduced. We choose one of the largest and most accurate networks, VGG16, and implement it on Xilinx Zynq ZC706 and Virtex VC707 boards, respectively. We achieve the top-5 accuracy of 86.25% using an equal distance non-uniform quantization method. It is estimated that the average performances are 316.23 GOP/s under 172-MHz working frequency on Xilinx ZC706 and 1250.21 GOP/s under 170-MHz working frequency on VC707, respectively. In brief, the proposed design outperforms the existing works significantly, in particular, surpassing related designs by more than two times in terms of resource efficiency.

Published in: IEEE Transactions on Circuits and Systems I: Regular Papers (Volume: 65 , Issue: 6, June 2018)

Page(s): 1941 - 1953

DOI: 10.1109/TCSI.2017.2767204

Date of Publication: 22 November 2017

Publisher: IEEE

ISSN Information:

Funding Agency:

SECTION I.
Introduction

Convolutional Neural Network (CNN) has been proven to be a powerful tool in domains of various tasks including object recognition [1], [2] and detection [3], [4], speech recognition [5], [6]. Real-time image recognition could be applied everywhere using CNN-like deep learning approaches in the near future. However, most of the CNN models are trained and implemented with software platforms due to the versatility. For mobility and privacy reasons, real-time applications requiring of high accuracy and low power image and voice recognitions based on CNNs should be able to be performed on local embedded processors.

CNN achieves satisfied recognition accuracy as long as the network is deep enough (a large number of layers), which makes it a deep Convolutional Neural Network (DCNN). Recent DCNNs, which consist of tens to hundreds of convolutional layers, have shown great promise in visual understanding [7].

Since existing systems with general purpose processors have not been optimized for DCNNs, the acceleration of DCNN should be carefully investigated for real-time embedded systems [8]. Most of the software based CNNs are implemented in GPUs [9], which have numerous execution units and large memory bandwidth to obtain high computational throughput. As a result, both the intensive computations of training and classification are shifted to GPUs. However, for local accelerators, GPU based implementations are infeasible due to limited hardware resources and tightly bounded energy consumption.

One of the solutions to provide required performance of DCNN within the restricted energy is to use hardware implementations such as ASIC and field programmable gate array (FPGA) [10]. Although the state-of-the-art CNN algorithms and designs are rapidly evolving, the low level operations stay the same, such as convolution and pooling. Convolutions in CNNs generally dominate the overall computational complexity and consume the major computation time and power in real implementations. Therefore, it is promising to develop efficient architectures for DCNNs in order to achieve high computation and energy efficiency.

There are three challenges of hardware based real-time high performance CNN implementations: 1) multiple and large-size feature maps in convolutional layers and the huge amount of parameters, both of which require large storage space; 2) the high complexity of convolutional computations, which greatly slows down the training and inference process; 3) the limited memory bandwidth, which incurs long data exchange delays. These factors hinder the real-time performance and the widespread deployment of DCNNs, particularly on resource constrained embedded systems.

Various hardware implementations of different CNN models have been proposed in recent years. Sankaradas *et al.* [11] presented a massively parallel coprocessor, whose functional units consist of parallel two-dimensional (2D) convolution primitives and programmable units, performing the pooling and non-linear functions in CNNs. Zhang *et al.* [12], proposed a hardware/software co-designed library to efficiently accelerate an entire CNN on FPGAs, which employed a uniformed convolutional matrix multiplication representation for both convolutional layers and fully connected layers. Chen *et al.* [13], Luo *et al.* [14], and Liu *et al.* [15] designed general accelerators for large scale CNNs and DNNs, with a special emphasis on the impact of memory on an accelerator. These three main strategies employed to optimize the memory systems are: 1) tiling and data reuse, for reduction of memory traffic; 2) storage buffer, for data reuse; 3) on-chip memory, for storage of all parameters [16].

Most of prior works focus on high level parallel architectures and efficient memory designs. However, these architectures seldomly reduce the number of computations significantly, especially in convolutions. In fact, 2D convolutions occupy more than 90% of the overall computation time [8]. The latency and energy consumed due to data access between external and internal memories lead to low computation throughput and large power consumption. Binary weights were employed in [17] and [18] to reduce both the computational complexity and the storage requirement at the cost of certain accuracy loss.

In this paper, efficient hardware architectures are proposed to process and store DCNN models. The main contributions of this work are:

- The theoretical derivation of 3-parallel fast finite impulse response (FIR) algorithm (FFA) is presented. Based on the 3-parallel FFA, an efficient 3-parallel Fast Convolution Unit (FCU) for convolutions with 3×3 kernels is developed. For convolutions with large convolutional kernels such as 5×5 and 7×7, we also derivate 5-parallel and 7-parallel FFAs and propose their corresponding FCUs.
- With the proposed FCU, multiple outputs can be computed in parallel, which improves the computation throughput. It is estimated that 33% of the multiplications are reduced with slightly increased add operations by the proposed 3-parallel FCU and the computational energy is reduced as well. Moreover, multiple FCUs are intelligently organized into an FCU array which enable efficient data reuse.
- A novel data reuse and storage scheme which avoids the transfer of intermediate data between internal memory and external memory is proposed. DRAM is utilized in a one-way transmission fashion, where only the raw input pictures and weights need to be read out and no data is required to be written back to DRAM. A further on-chip memory reuse scheme is proposed to reduce memory requirement.
- The equal distance non-uniform quantization (ENQ) method is introduced to our quantization flow, which contributes to the reduction of data width.
- An overall hardware architecture which contains scalable computational logics and storage resources is proposed. Three methods of parallel processing are implemented by the proposed architecture, which considerably exploit the parallel computations of DCNNs.

The rest of this paper is organized as follows:

In Section II, we introduce the background of CNN and implementation model. In Section III, the fast convolution algorithm is described in detail and the corresponding fast convolution unit is proposed. The proposed ENQ scheme is introduced in Section V. The novel storage scheme and computation flow are proposed and described in detail in Section IV. Three methods of parallel processing along with the computation and storage architectures are presented in Section VI. We analyse the architecture efficiency in aspects of computing, storage, and energy in Section VII and compare the implementation results with previous works in Section VIII.

SECTION II. Background

A CNN can be viewed as the combination of a feature extractor and a category classifier. Three architectural ideas are introduced to ensure the invariance of some degree of shift, scale, and distortion: 1) local receptive fields; 2) shared kernel weights; 3) spatial or temporal sub-sampling (pooling) [5]. CNNs are usually composed of three main different types of layers: convolutional (CONV) layers, pooling layers, and fully connected (FC) layers. Some new types of layers are developed such as normalization and dropout layers. A CNN model usually has a feed-forward design, where each layer takes the output of a preceding layer as its input and produces the results for the next layer. Normally, pooling layers follow convolutional layers and fully connected layers are the last few layers. Fig. 1 shows an example of CNN design.

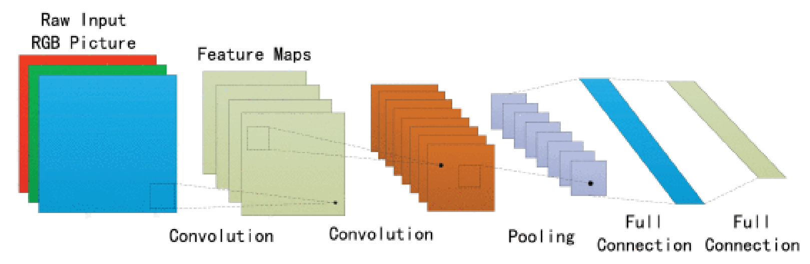


Fig. 1.
Design of a CNN model.

A. Convolutional Layers

Convolutional layers are the major parts of a CNN. Each output pixel is connected to only a local region of the input layer and the extent of this connectivity is called the receptive field. The connections are local in space (along length and width), but always extend along the entire channel of the input layer. The receptive fields are overlapped both along the height and width by a certain stride (usually 1) in an input feature map and the kernel weights are shared with them. The convolution in CNN is a 2D operation, where shared kernel weights are multiplied with the corresponding receptive field in an element by element way. These products are summed together with an additional bias.

Usually, an input layer contains multiple feature maps. The convolution results of all input feature maps are added together to get an output feature map. Thus the kernels are extended to 3 dimensions, where each 2D kernel corresponds to an input feature map. The pixel y at location (x, y) in output feature map n is given by:

$$\begin{aligned} sum &= \sum_{i=0}^{N_i-1}\sum_{j=0}^{Ky-1}\sum_{k=0}^{Kx-1} w^{(n)}[i][j][k]\times in[i][x+j][y+k],\\ y^{(n)}[x][y] &= f(sum+b[n]),\end{aligned}$$

View Source

where $w^{(n)}$ and N_i represent the kernel weights and the number of input feature maps, respectively. Ky and Kx denote the height and width of the convolutional kernel, $b[n]$ is the bias added to output feature map n , respectively. f represents the activation function, which could be **sigmoid, tanh or ReLU**.

B. Pooling Layers

Pooling layers aim at simplifying and smoothing the precedent feature maps. Usually, a pooling layer follows a convolutional layer. Two types of pooling operations are commonly employed, which are average and max pooling. Average pooling computes the mean of a small local field in each input feature map, while max pooling picks the max value of the local field to output a pixel. For a pooling layer, the number of output feature maps equals to that of input feature maps.

C. Fully Connected Layers

Fully connected layers are placed at the end of a CNN design and perform as classifiers. They are usually flattened to one dimension and act in the form of regular neural networks, where every neuron has full connections to all neurons in the previous layer.

D. Implementation Model

The VGG model [19], proposed by Simonyan and Zissermanachieved, won the second place in image classification task of ILSVRC 2014. VGG models are a series of networks with deep layers and use very small (3×3) convolutional kernels. It was shown that a significant improvement on the classification can be achieved by pushing the depth to 16–19 layers. In general case, VGG models consist of multiple convolutional layers, 4 max-pooling layers and 3 fully connected layers.

Only 3×3 kernels are employed in VGG model so that fewer weights are needed and higher accuracy is obtained. VGG model can branch out to VGG11, VGG16, and VGG19 depending on the total number of layers. In this paper, the VGG16 model is considered due to its relative balance between complexity and accuracy.

SECTION III.
Fast Convolution Algorithm

A. Parallel Fast FIR Algorithm

An FIR filter with N taps can be expressed as:

$$\begin{aligned} y(n) =&h(n) * x(n) = \sum_{i=0}^{N-1} h(i)x(n-i), \\ & n = 0, 1, 2, \cdots, \infty, \end{aligned} \tag{1}$$

View Source

where $x(n)$ is an infinite input sequence and $h(n)$ contains the coefficients of an N -length FIR filter. The convolution shown in Eq. (1) can be expressed in z domain:

$$Y(z) = H(z)X(z) = \sum_{n=0}^{N-1} h(n)z^{-n} \sum_{n=0}^{\infty} x(n)z^{-n}. \tag{2}$$

View Source

If the input sequence $x(n)$ is decomposed into even and odd indexed samples, we have $X(z) = X_0 + z^{-1}X_1$. Similarly, $H(z) = H_0 + z^{-1}H_1$ and the output:

$$Y(z) = Y_0 + z^{-1}Y_1 = (X_0 + z^{-1}X_1)(H_0 + z^{-1}H_1), \tag{3}$$

View Source

where

$$\begin{aligned} Y_0 =& H_0X_0+z^{-2}H_1X_1, \\ Y_1 =& H_1X_0+H_0X_1. \end{aligned} \tag{4}$$

View Source

Based on the fast FIR algorithm (FFA) [20], Y_0 and Y_1 can be expressed in the form of 2-parallel FFA,

$$\begin{aligned} Y_0 =& H_0X_0+z^{-2}H_1X_1, \\ Y_1 =& (H_0+H_1)(X_0+X_1)-H_0X_0-H_1X_1. \end{aligned} \tag{5}$$

View Source

Similarly, the input sequence $x(n)$ and tap coefficients $h(n)$ can be decomposed into three parts as follows:

$$\begin{aligned} Y =& Y_0+z^{-1}Y_1+z^{-2}Y_2 \\ =& (X_0+z^{-1}X_1+z^{-2}X_2)(H_0+z^{-1}H_1+z^{-2}H_2) \\ =& (X_0+z^{-1}V)(H_0+z^{-1}W), \end{aligned} \tag{6}$$

View Source

where $V=X_1+z^{-1}X_2$ and $W=H_1+z^{-1}H_2$. The last equation of Eq. (6) and its intermediate term VW are both in the 2-parallel FIR filter form and can be computed by employing Eq. (5) recursively [20]. Thus, the 3-parallel FFA is computed as follows:

$$\begin{aligned} Y_0 =& H_0X_0-z^{-3}H_2X_2+z^{-3}[(H_1+H_2)(X_1+X_2)-H_1X_1], \\ Y_1 =& [(H_0v+H_1)(X_0+X_1)-H_1X_1]-[H_0X_0-z^{-3}H_2X_2], \\ Y_2 =& [(H_0+H_1+H_2)(X_0+X_1+X_2)] \\ & -[(H_0+H_1)(X_0+X_1)-H_1X_1] \\ & -[(H_1+H_2)(X_1+X_2)-H_1X_1]. \end{aligned} \tag{7}$$

View Source

For parallel FFAs of larger sizes of prime numbers (*e.g.*, 5 and 7 parallel FFAs), we have already derived their expressions, but do not put them down in this paper because the derivation process is relatively complex compared to 3 parallel FFA. It should be noted that large kernels (*e.g.*, 5×5 and 7×7) are indeed employed in some CNN models such as [7] and [21]. Parallel FFAs with sizes of composite numbers can be obtained by applying cascading short term FFAs [20], [22].

Parallel FFAs are essentially parallel computing algorithms. A large size filter is decomposed into several small sub-filters and each performs short convolutions. For a filter of size P , where P is a prime number such as 3 or 5, the filter can be decomposed into P sub-filters. For the presented parallel FFAs, the convolution of each sub-filter becomes a multiplication. These partial convolutional results are added with specific combinations to compute several output results in parallel.

Reference [23] introduced Winograd's minimal filtering algorithms to minimize multiplications in small size FIR filters. However, more additions and complex pre-computations are required, which is unfriendly to hardware implementation. Moreover, it is difficult to use Winograd's minimal filtering algorithms to derive the expressions of large tap filters with fewer additions.

B. Fast Convolution Unit

For convolution with a $k\times k$ kernel in a CNN, the 2D convolution is decomposed into k 1D convolutions and each 1D convolution can be implemented by a k -tap FIR filter, where the tap coefficients are the corresponding kernel weights. Based on the parallel FFA, a Fast Convolutional Unit (FCU) is proposed to perform the convolution in a CNN.

Take the 3×3 kernel as an example, the 3-parallel FCU architecture is shown in Fig. 2, where the coefficients h_0 , h_1 , h_2 denote the kernel weights, x_0 , x_1 , x_2 and y_0 , y_1 , y_2 are inputs and outputs, respectively. They all belong to one row or column in a 2D field.

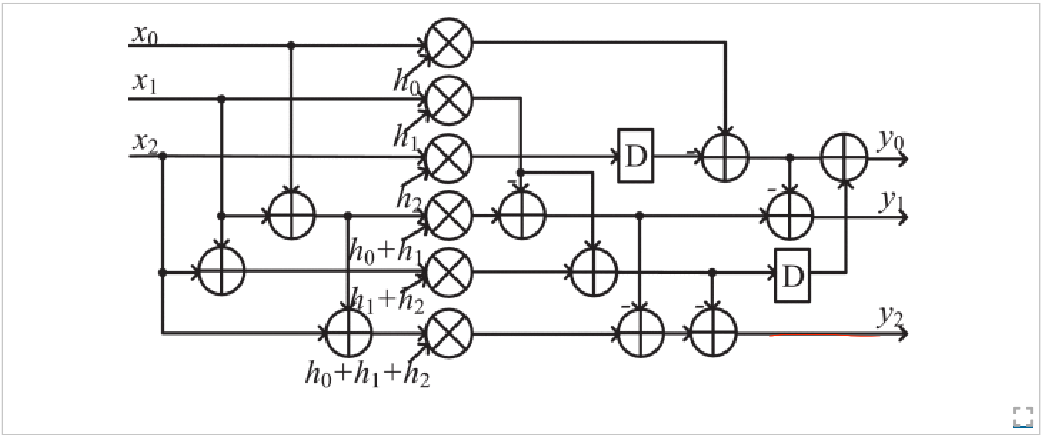


Fig. 2.
Architecture of the 3-parallel FCU.

It is noted that the kernel weights must be in the reverse order in each FCU, since in convolution the tap coefficients are reversed and followed by element-wise multiplications. For example, if the kernel weights in one row of a 3×3 kernel are in the order of k_0 , k_1 , k_2 , then in the 3-parallel FCU, $h_0 = k_2$, $h_1 = k_1$, $h_2 = k_0$.

In order to finish a 2D convolution, k identical FCUs are employed. The outputs of all FCUs are added together. For example, when $k = 3$, three 3-parallel FUCs are needed to perform the 2D convolution. For $j = 0, 1, 2$, let $y_{j,0}$, $y_{j,1}$, $y_{j,2}$ denote the outputs of the j -th FCU. The m -th output of the 2D convolution is $y_{0,m} + y_{1,m} + y_{2,m}$ for $m = 0, 1, 2$. The 2D convolution with three 3-parallel FCUs are illustrated in Fig. 3.

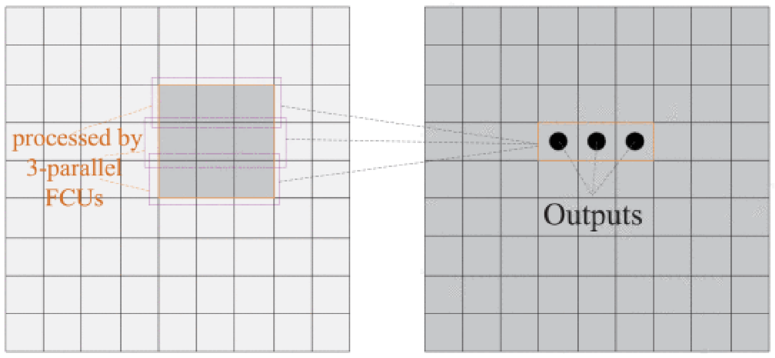
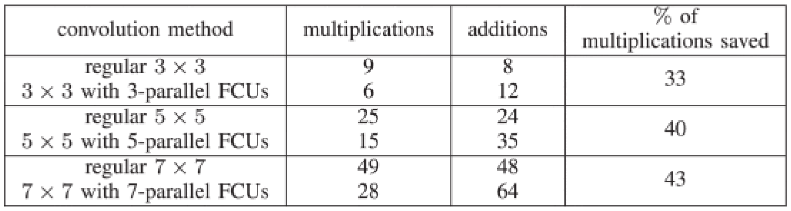


Fig. 3.
Illustration of convolution with three 3-parallel FCUs.

From the 3-parallel FFA and 3-parallel FCU, it is estimated that 33% multiplications can be saved with some extra additions.

With large convolutions such as 5×5 and 7×7 , based on 5 and 7 parallel FFAs, the corresponding 5 and 7 parallel FCUs are designed and employed to perform the corresponding convolutions. The comparisons of arithmetic operations are listed in Table I. Because convolutions with FCUs are feedforward designs, the pre-additions and post-additions can be pipelined to increase the clock frequency.

TABLE I Comparisons of Arithmetic Operations



convolution method	multiplications	additions	% of multiplications saved
regular 3×3	9	8	33
3×3 with 3-parallel FCUs	6	12	
regular 5×5	25	24	40
5×5 with 5-parallel FCUs	15	35	
regular 7×7	49	48	43
7×7 with 7-parallel FCUs	28	64	

SECTION IV. Memory Efficient Computation Flow

A. Inter-Layer Partial Storage and Intra-Layer Ping-Pong Reuse Scheme

Due to limited on-chip memory resources, tiling is commonly used for most hardware implementations of CNNs. Row (or column) tiling [24] and channel tiling [13] are often applied. Tiling is essentially a split of block data from large chunks to small ones that can be stored on chip. However, data tiling does not avoid intermediate data exchanges between internal and external memories in previous architectures. The data transmission latency is large and the throughput is constrained by the interface bandwidth. In this paper, we propose a novel inter-layer partial storage and intra-layer ping-pong reuse scheme. This scheme makes the best use of on-chip memory resources and does not need to send intermediate data to external memories such as DRAM.

Each input feature map of some convolutional layers is tiled by factor T_r in row and the T_r rows of pixel data are stored in a specific on-chip block RAM, including the raw input picture layer which contains three feature maps of RGB channels and their tiled T_r row data is stored in raw picture RAM. The convolutional kernel size is $k \times k$ and $T_r \geq k$. From the calculation, $T_r - k + 1$ rows of pixel data will be computed assuming the stride is 1. Each of these block RAMs has two segments, both of which stores $\frac{T_r}{2}$ rows of pixel data. One segment is for buffering the fresh new calculated $T_r - k + 1$ row data and the other is to store the previous $T_r - k + 1$ row data. These two segments are utilized in a ping-pong manner and T_r is calculated as follows:

$$\begin{aligned} 2(T_r - k + 1) &= T_r, \\ T_r &= 2k - 2. \end{aligned} \tag{8}$$

View Source

Reference [25] proposed an efficient dataflow across convolutional layers, which employed a pyramid-shaped multi-layer sliding window to enable effective on-chip caching during CNN evaluation. However, it only tested on fusing the first five convolutional layers of the VGGNet-E network and the on-chip memory requirements grow exponentially as the network goes deeper with their method.

To further make reuse of the on-chip memories, the block RAM sizes of some convolutional layers should be the same and the number of rows stored in such layers is identical.

Not all block RAMs need to store T_r rows of the corresponding layers. Once a convolutional layer arranged before a pooling layer is computed, there is no need to store the previous rows of such layer. Such convolutional layer is only allocated half size of the block RAM compared to other convolutional layers and the block RAM just stores $\frac{T_r}{2}$ rows of pixel data.

Before the description of computation flow, we signify three status signals to represent the storage conditions of the block RAMs that store T_r row data. The three status signals are “empty”, “half”, and “full”. Each of these block RAMs has two segments, both of which has a label signal which can be labelled as “1” or “0”. The representation of these signals are expressed as follows:

- “1” indicates that a segment has stored the computed pixel data which will participate in the further convolutions.

- "0" represents that a segment has not received any computed pixel data yet or the segment stores the data which was used for computations twice and can be overwritten.
- "empty" indicates that the two segments of the block RAM are labelled as "00" and pixel data can be written to either of the segments.
- "half" means the two segments are labelled as "10" or "01". The corresponding block RAM has stored $\frac{T_r}{2}$ rows of pixel data and only the segment labelled as "0" can store newly computed pixel results.
- "full" represents that the two segments of the block RAM are labelled as "11", which has already stored T_r row data in a convolutional layer and pixel data can be sent to convolution operations immediately.

The computation flow from layer to layer is described as follows and Fig. 4 shows an example of the data inter-layer partial storage and intra-layer ping-pong reuse scheme.

Step 1: Load raw input picture data to fill up the raw picture RAM and set the status signal as "full". Set the signals of all other block RAMs as "empty".

Step 2: Once the status signal of an input block RAM C_i is "full", the stored pixel data is sent to computation logics. Then the output $\frac{T_r}{2}$ row data is written to one segment labelled as "0" of the output block RAM C_{i+1} . In the mean time the computations are being processed, the raw picture RAM loads the next $\frac{T_r}{2}$ rows of raw input pictures if the status signal is not "full". Weight Buffer (WB) receives weights for next convolutions from external memories meanwhile.

Step 3: When the convolutions are completed, the segment label of output block RAM C_{i+1} turns from "0" to "1". The signal of block RAM C_{i+1} is set as "full" or "half" depending on whether the other segment is labelled as "1" or "0". Meanwhile, in input block RAM C_i , the label of the segment which stores the row data computed twice is set as "0". Then turn to **Step 2**.

Step 4: Before resuming the convolutions, if status signals of the raw picture RAM and an input block RAM are both "full", the input block RAM is used to operate convolutions in priority.

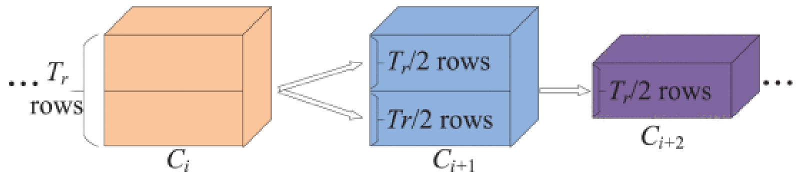


Fig. 4.
An illustration of data inter-layer partial storage and intra-layer ping-pong reuse.

For VGG16 model, the convolutional kernel size is 3×3 and $T_r = 4$. Fig. 5 shows the data flow of VGG16 model with the proposed inter-layer partial architecture. It is shown that all layers are allocated a corresponding block of attached memories.

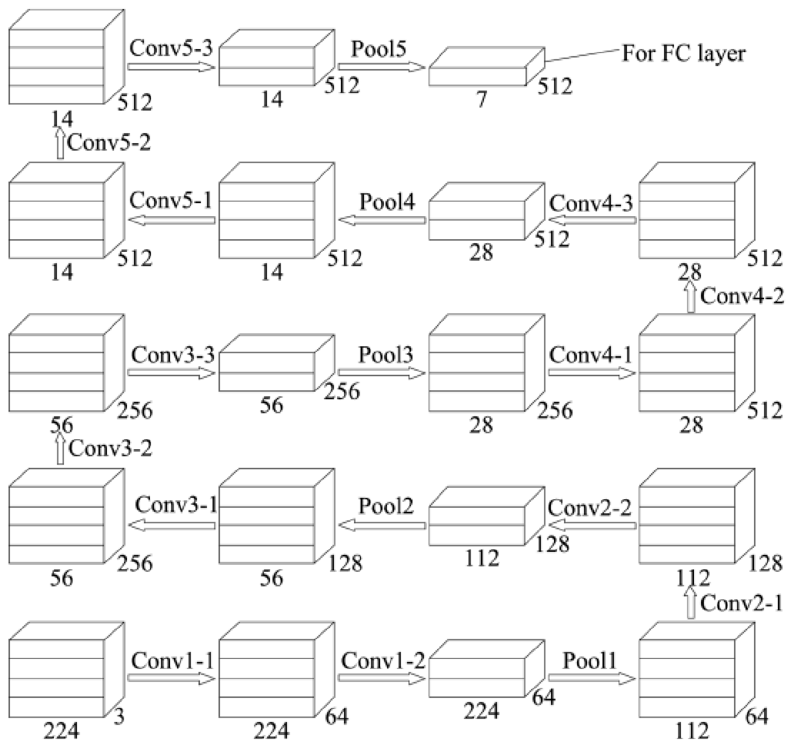


Fig. 5.
Data flow of VGG16 with partial-inter-layer storage architecture.

Fully Connected (FC) layers are usually arranged after convolutional layers and mainly consist of matrix multiplications. With the proposed inter-layer partial storage and intra-layer ping-pong reuse scheme, the matrix multiplications of FC layers are split into sub matrix multiplications. In Fig. 5, for example of VGG16, the Pool5 layer is connected to a FC layer, which is split from $7 \times 7 \times 512 \times 4096$ to $7 \times 512 \times 4096$. While the sub matrix multiplications are being processed, the previous operations, *e.g.*, convolution and pooling, can be operated simultaneously.

B. Further On-Chip Memory Reuse

The memory reuse can be further exploited even if we have already taken advantage of on-chip memories. The memory reuse scheme can be arranged in two aspects: 1) The full reuse of RAMs belonging to convolutional layers which are arranged adjacently before pooling layers; 2) The partial reuse of RAMs belonging to some layers with the same storage design. These two further memory reuse schemes require particular pixels quantized to be the same bits. Therefore, in the ENQ method, pixel data of particular layers should be consistently extended to the maximum bits of the pixel data of these layers.

The data flow of RAM partial reuse scheme is roughly presented in Fig. 6. $r_s^{(i)}$ denotes a segment of the i -th RAM storing $\frac{1}{2}T_r$ rows of pixel data. The subscript s stands for the segment label which is either “0” or “1”. The heuristic idea is that the computed $\frac{1}{2}T_r$ row data of layer i can be stored in a segment of RAM j instead of RAM i for layer i . Therefore, only if the block RAMs have the same storage design, they can be reused. Once two segments of 1 or 2 block RAMs are labelled as “1” and both the segments store the row data of the same layer, the computation is triggered and the output $\frac{1}{2}T_r$ rows of pixel data are written to a segment which is labelled as “0”. When the computation is completed, one of the segments whose stored pixel data is computed twice is labelled as “0” and can be overwritten in further computations.

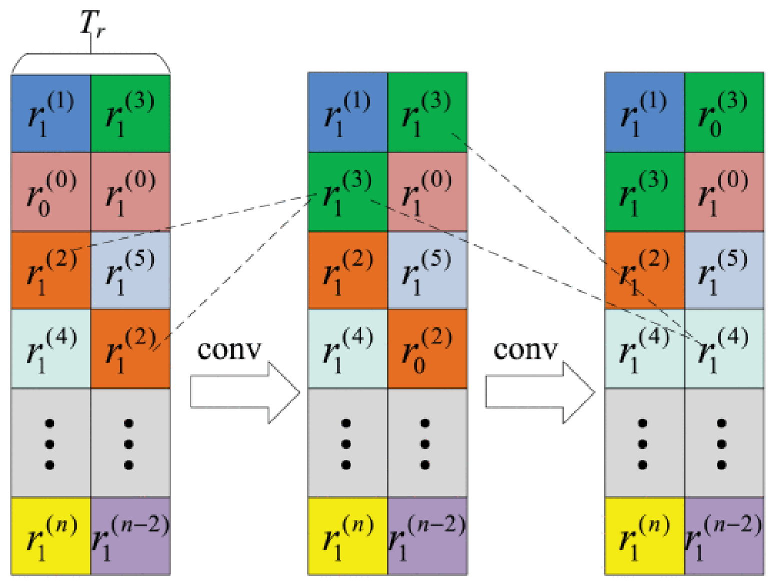


Fig. 6.
The data flow of further on-chip memory reuse.

For VGG16 model, RAMs for corresponding layers that can be reused in the two different memory reuse schemes are marked in Fig. 7.

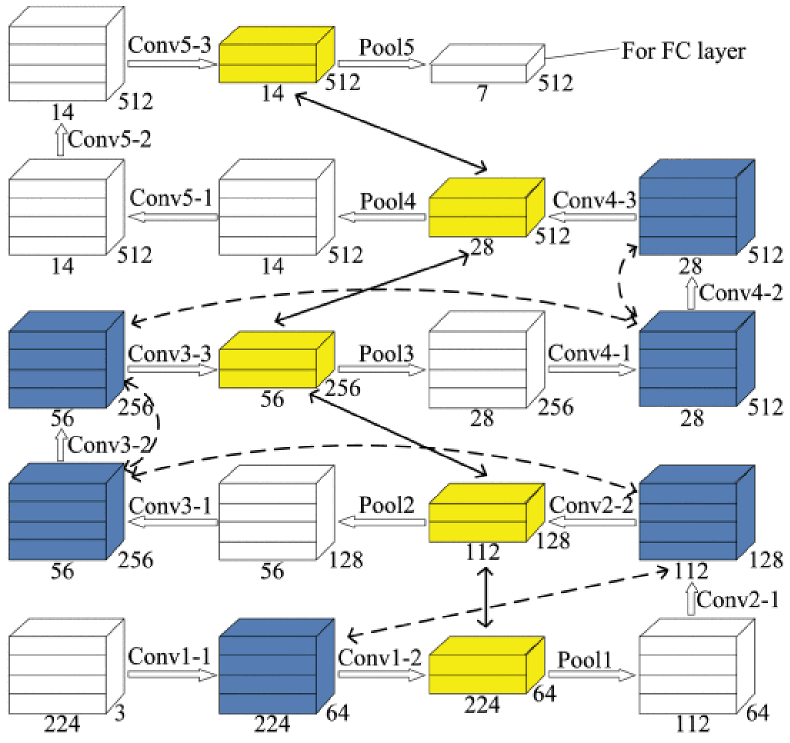


Fig. 7.
Further reuse of on-chip memories for architecture of VGG16 data flow.

The fully reused block RAMs are linked by solid arrows. These convolutional layers share the same block RAM, size of which is determined by the largest one. By coincidence, in VGG16 these block RAMs are of the same size except for the last one. The partial reuse among block RAMs are linked with dash arrows. On the basis of inter-layer partial storage and intra-layer ping-pong reuse scheme for VGG16 implementation, the further on-chip memory reuse method can further reduce about 20% of block memories without any overhead.

For VGG16 implementation, in order to avoid the whole intermediate pixel data transfer with regular layer-wise implementation, the number of feature map pixels stored on chip are 6.4M. With the memory-efficient computation flow, the required feature map pixels stored on chip are reduced to 463.5k. Our design reduces memory requirement by 14 times and the intermediate pixels are no longer required to be sent to off-chip DRAM.

Our proposed inter-layer partial storage and intra-layer ping-pong reuse scheme are especially hardware friendly to other CNNs with residual blocks [1]. The residual block function can be roughly expressed as: $H(x) = F(x) + x$. $F(x)$ is the residual block and consists of several convolutional layers. The inputs and outputs of residual blocks are added together, which requires that the input feature maps must be preserved in DRAM and read out later in the regular layer-wise process. However, with the proposed method, we store partial row data of all layers in on-chip RAMs and the row data can be directly fetched and added to the outputs of residual blocks.

SECTION V. Quantization Method

In CNN implementation, quantization is necessary to reduce the computation and storage bits while preserving the recognition accuracy. Previous works focus on the quantization of kernel weights and the quantization of pixel data is rarely discussed. The equal distance quantization (ENQ) method is proposed in this section.

It was shown in [26] that the distributions of numerical values of both pixels and weights in most layers are roughly Gaussian distribution. For VGG16 model, the distribution of pixel values in each convolutional layer is Gaussian-like as well [27].

A. Quantizaition Flow

The quantization method is to a map pixel values to a set of quantization points (QPs) and each QP corresponds to a fixed-point value. $\{P_0, P_1, \cdots, P_{N-1}\}$ denote a set of QPs with N elements and V_i denotes the corresponding fixed-point value associated with P_i $(i = 0, 1, \cdots, N-1)$. A pixel value s is quantized to a QP:

$$s \rightarrow P_k, \quad k = 0, 1, \cdots, N-1. \tag{9}$$

View Source

It requires $K = \lceil \log_2 N \rceil$ bits to store a quantized pixel value.

From [27], it can be concluded that most pixel values are relatively small in a layer. It is reasonable to assign more QPs to represent the smaller pixels without incurring obvious degradation of accuracy. The proposed quantization scheme is described as follows:

- For uniform quantization, *i.e.*, each pixel is quantized to q bits (2^q QPs in total) and each weight is quantized to q_w bits. All pixel values are non-negative because we store pixels after the ReLU operation. Let F represent the number of fractional bits in the q -bits uniform quantization. Therefore, $V_i = i2^{-F}, i = 0, 1, \cdots, 2^q - 1$. A and A' represent the accuracy with floating and q -bit uniform quantization, respectively. The minimal value of q (denoted as q') is calculated so that $A - A' \leqslant \delta$, where δ is a small positive number determined by the corresponding data set and application.

- For pixels within layer i , the proposed ENQ employs an E_i -bit nonuniform quantization scheme, where $E_i \leqslant q_m$ and q_m is the uniform quantization bits in the previous step. For the proposed E_i -bit

ENQ scheme, there are 2^{E_i} QPs, $P_{i,0}$, $P_{i,1}, \cdots, P_{i,2^{E_i}-1}$, where $P_{i,k}$ corresponds to the fixed-point value $V_{i,k} = k2^{q_m - E_i}2^{-F}$. Here, $k = 0, 1, \cdots, 2^{E_i} - 1$. Supposing the magnitude of an pixel is x , the ENQ scheme quantizes it to $\lfloor \frac{x}{2^{q_m - E_i}} \rfloor$ if $x \leqslant 2^{q_m} - 1$. Otherwise, x is quantized to $2^{E_i} - 1$. Note that each pixel value within layer i is stored using E_i bits. When a pixel is required to participate in the convolutions in the next layer, it should be converted to q_m -bit based on the relationship between QPs and fixed-point values. In this work, an exhaustive search is employed to find the minimal E_i for each layer i such that the resulting accuracy is close to A' .

B. Numerical Results

We first perform the uniform quantization of VGG16 model and the weight parameters are all quantized to 8 bits in the uniform quantization step. As shown in Table III, the accuracy of the full precision is 88.5%. The same bit-width quantization is utilized for all layers without fine-tuning and 12 bits are enough for the uniform quantization of all layers.

The numbers of quantization bits generated by ENQ scheme are shown in Table II, which includes several combinations of E_i and the corresponding accuracy. We adopt the last indexed quantization scheme in Table II with top5 accuracy of 86.25% and q_m is 12 bits.

TABLE II ENQ Result of VGG16 Model

Table II- ENQ Result of VGG16 Model

TABLE III Uniform Quantization of VGG16

Table III- Uniform Quantization of VGG16

In our experimental case of VGG16 model, on-chip memories are further saved by 54.3% when the inter-layer partial storage and intra-layer ping-pong reuse scheme is employed. If the further on-chip memory reuse method is utilized, on-chip memories can be further saved by 47.0%. Both cases remain other conditions unchanged except that ENQ is employed compared to uniform quantization method.

SECTION VI. Efficient Hardware Architecture for DCNN

For a DCNN, each convolutional layer contains multiple feature maps and convolutions occupy more than 90% of the overall computation time. In order to improve the performance, the degree of computing parallelism should be increased as much as possible. However, it takes careful consideration to decide how to process in parallel. When it comes to parallel processing, data reuse should be employed to reduce the number of redundant memory read or write. Both feature maps and parameters can be reused but in different ways depending on how to perform the computations in parallel. For previous hardware architectures of CNN, two common methods are applied to achieve parallel data processing. One method is data processing in the direction of one dimension which is along length or width on a feature map and parameters are reused in this way. The other is data processing along channel in a layer, where feature maps are reused.

In this paper, both of these two methods are intelligently combined together to accelerate the DCNN computations and improve the throughput. Moreover, due to the intrinsic pipeline property of memory-efficient computation flow introduced in Section IV, we introduce a third parallelism method for the first time, the inter-layer parallel processing.

A. Parallel Processing Along Row & Column

With the proposed FCU architecture, the convolution patterns are quite different from the regular ones. Pixel data in each row or column is reconstructed as connection and interdependence, which are explained as follows:

- Connection means that k pixels are packaged and sent to a k parallel FCU at a time and k output pixels are calculated. In the regular k -taps convolution form, k pixels are multiplied with corresponding coefficients and added together to obtain one output pixel. Here, the size of the convolutional kernel is $k \times k$.

- Interdependence means that once the former package of k pixels in a row on a feature map are consumed, the adjacent k pixels must follow into the same FCU immediately and the computing process repeats until all pixels of a row are processed.

By organizing $k \times (k-1)$ FCUs into a computing array, we construct a processing unit (PU) as a basic structure to perform the convolutions. This architecture is illustrated in Fig. 8, where the data reuse is achieved in the following two ways:

- Kernel weight, W_i , is shared among $k-1$ FCUs in the i -th row of the computing array of PU for $i=0,1,\cdots,k-1$.

- $2k-2$ rows of pixel data as shown in Fig. 8 are partially shared among certain FCUs. The results of all k FCUs in each array column are summed together to obtain the intermediate data of a row on an output feature map.

Fig. 8. - Processing Unit (PU) composed of FCU arrays.

Fig. 8.
Processing Unit (PU) composed of FCU arrays.

In VGG16 model, a PU aims at processing pixel data along rows on a feature map, which contains $3 \times 2 = 6$ 3 -parallel FCUs. Each input and output of an FCU is a package of 3 pixels. Thus the 3-parallel PU can process 12 pixels and output 6 pixels in parallel.

The computing process of a 3-parallel PU is shown in Fig. 9. With each proposed PU, 12 pixels (4×3) from four adjacent rows on an input feature map are used in parallel. Therefore, four rows on an input feature map can be processed and two output rows are obtained in parallel by the mean time.

Fig. 9. - The computing process of 3×3 convolutions realized by a 3-parallel PU.

Fig. 9.
The computing process of 3×3 convolutions realized by a 3-parallel PU.

B. Parallel Processing Along Channel

The parallel processing along channel is a multiple-to-multiple pattern of feature maps. To improve the throughput of the design, the degree of computing parallelism should be exploited as much as possible. The multiple-to-multiple pattern employed in our processor is described in Fig. 10.

Fig. 10. - The multiple-to-multiple pattern of parallel processing along channel.

Fig. 10.
The multiple-to-multiple pattern of parallel processing along channel.

The diagram shows that multiple input feature maps are processed in parallel and multiple output feature maps are obtained.

To achieve the goal of multiple-to-multiple pattern processing, an m-to-m processor which consists of all the computation logics and storage resources is proposed. The m-to-m processor contains Complex Processor (CP), Pixel RAM (PRAM), Attached RAM (ARAM), Weight Buffer (WB), and DRAM.

CPs are cores of the m-to-m processor. All computation logics are integrated in the CPs and data communication are the most frequent between memories and CPs. A CP is composed of Bit-width Converter (BC), Processing Unit (PU), Adder Compressor, ReLU module, and Max Pooling unit. The detailed architecture of CP is shown in Fig. 11 and described as follows:

- **Bit-width Converters (BCs)** act to convert the bit width of pixel data between computations and storage as explained in the ENQ method.

- **Processing Units (PUs)** are FCU arrays and each PU operates the 2D convolution of an input feature map. T PUs are employed in a CP and partial rows of T input feature maps are processed in parallel. In an m-to-m processor, there arranged PT PUs and for the efficient usage of hardware resources, T equals P .

- **Adder Compressor** is employed to implement arithmetic and digital signal processing (DSP) circuits for low power and high performance applications [28]. In order to reduce the critical path, we employ 4–2 compressor instead of adder trees. The 4–2 Compressor has 5 inputs A, B, C, D and Cin to generate 3 outputs Sum, Carry and Cout as shown in Fig. 12(a). The input Cin is the output from a previous lower compressor and the Cout output is for the compressor in the next stage. The regular approach to implement 4–2 compressor is with 2 full adders connected serially and the enhanced version in [28] which are employed in our design. This enhanced compressor utilizes the XOR-XNOR module and the transmission gate version of the MUX module as shown in Fig. 12(b). If the sums of Adder Compressor are the final results of feature maps, then they are sent to ReLU modules. Otherwise, they are sent to ARAMs as intermediate results.

- **ReLU** modules receive biases from WBs and add them to the output sums of Adder Compressors. Then they perform as the max selectors between zero and corresponding sums and send the final results to PRAMs.

- **Max Pooling** units perform the 2×2 pooling function with pixel data from ARAMs and send the results to corresponding PRAMs.

Fig. 11. - The architecture of a Complex Processor (CP).

Fig. 11.
The architecture of a Complex Processor (CP).

Fig. 12. - (a) 4–2 adder compressor. (b) Enhanced architecture with the XOR-XNOR and MUX modules of a 4–2 compressor.

Fig. 12.
(a) 4–2 adder compressor. (b) Enhanced architecture with the XOR-XNOR and MUX modules of a 4–2 compressor.

The overall architecture of the m-to-m processor is shown in Fig. 13. Each component is described as follows:

- **Complex Processors (CPs)** are the main computation units employed to realize the multiple-to-multiple pattern of parallel processing. Suppose P CPs are employed and each CP outputs the partial rows of an output feature map. Apart from convolutions, bit-width conversion, ReLU, and max pooling are also performed by a CP. CPs exchange pixel and weight data with pixel RAMs, Attached RAMs, and Weight Buffers.

- **Pixel RAMs (PRAMs)** are the combinations of block RAMs which store the partial rows of pixel data of all layers. PRAMs exchange pixel data with CPs as described above. Only the $\mathrm{PRAM}_{\mathrm{in}}$ need to receive input raw picture data from off-chip DRAM and other PRAMs have no data access with external DRAM, so they are highly efficient due to low latency.

- **Attached RAMs (ARAMs)** perform a similar function as PRAMs but they buffer intermediate results and send pixel data to max pooling module. They do not need data access with DRAMs either.

- **Weight Buffers (WBs)** are the channels which send kernel weights and biases from off-chip DRAM to CPs. WBs are used because the weights are too many for on-chip memory to store all at once. While the current step of convolution is being performed, the kernel weights required for the following convolutions are sent to weight buffers meanwhile.

Fig. 13. - Overall architecture of the m-to-m processor.

Fig. 13.
Overall architecture of the m-to-m processor.

C. Inter-Layer Parallel Processing

Since the computation dataflow presented with our architecture is quite different from the traditional layer-wise one, the inter-layer parallel processing can be conducted. In the traditional feedforward layer-wise dataflow, the computation of a convolutional layer can not start until its previous convolutional layer is completedly processed. Then the previous layer can be dropped out. However, the computation is restricted between two adjacent convolutional layers and the process of a convolutional layer must wait until all the previous layers are processed.

With our architecture, the convolutions are not constrained between two layers. They are split into segments and extend from the first layer to the last layer. So computations of different layers can be under taken at the same time.

For instance, we realize the inter-layer parallel processing of data loading of input layer, computations of convolutional layers and fully connected layers, and max pooling of pooling layers. The detailed parallel process is explained in the following steps:

- Once the T_r rows of the input raw picture layer are calculated, then $\frac{T_r}{2}$ new rows of pixel data begin to be loaded from the off-chip DRAM and overwrite the segment labelled as '0'. Therefore the loading time of the input raw picture layer is saved.

- When it begins to calculate the pooling layer, the convolution operations of its former layers including the input raw picture layer can be trigged simultaneously. Under usual circumstances convolution time is much longer than that of pooling and the pooling time can be saved.

- When the pooling operation of the last pooling layer is completed, only one row is outputted. For instance, in VGG16 model it is $1 \times 7 \times 512$. These partial final results are fed to the fully connected layer. The operation of fully connected layer is essentially matrix multiplications. The whole matrix multiplications are divided into several partial multiplications and additions (MACs). Each part can be calculated simultaneously with other layer operations such as convolutions and data loading. Due to the long interval time between two partial rows of the last pooling layer, the computation time of fully connected layer can be saved.

We combine the above three parallel processes together to obtain the max degree of parallelism and improve the throughput of our design considerably.

The critical path is restricted by the operations of convolution in our architecture. Due to the feedforward hardware design of convolutional layer, multiple stages of pipelines can be inserted to reduce the critical path. In our case, a three-stage pipeline is employed in the design.

SECTION VII. Efficiency Analysis

A. Computing Efficiency Analysis

The massive computations are intensive in convolutional layers and the performance of the proposed system is evaluated in this section.

For CONV layer, the required number of computation cycles of layer i with the proposed architecture can be calculated by the following formula:

$$N_{cycles}^{CONV_i} = \left\lceil \frac{L_{in}^i}{k} \right\rceil \times \left\lceil \frac{n_{in}^i}{P} \right\rceil \times \left\lceil \frac{n_{out}^i}{P} \right\rceil, \quad i = 1, 2, \ldots, \tag{10}$$

View Source

where P denotes the degree of parallelism, k is the kernel size, L_{in}^i represents the length of input feature map, and n_{in}^i and n_{out}^i are the numbers of feature maps (channels) of input and output layers repectively.

The above equation handles the computations of T_r rows in input layer to $\frac{T_r}{2}$ rows in output layer. The total computation cycles of a whole input layer to an output layer is shown in the following equation:

$$N_{cycles\_total}^{CONV_i} = N_{cycles}^{CONV_i} \times \left\lceil \frac{2W_{in}^i}{T_r} \right\rceil, \tag{11}$$

View Source

where T_r denotes the row-tiling factor, W_{in}^i and W_{out}^i represent the width of input feature map and output feature map, respectively.

Since pixels are stored in PRAMs, no pixel needs to be sent to off-chip DRAM, so there is no loading and unloading time of feature maps.

For each feature map in an input convolutional layer, it will be reused P times. It is emphasized that the same reused input feature map for different output feature maps corresponds to different kernels. In our architecture, each PU is fed with $k \times k$ weights and we employ P CPs, each of which contains P PUs, so the total number of weights fed to overall architecture is $k^2 P^2$ every *computing phase* (*computing phase* is defined as the computing process of P feature maps). The kernel weights do not need to be updated until a *computing phase* is completed, so the minimum clock cycles during which the same batch of kernel weights can stay unchanged are $min\left\{\left\lceil \frac{L_{in}^i}{k} \right\rceil\right\}$. The clock frequency is set to f and each weight is quantized to Q bits. The minimum refresh frequency of kernel weights is $f/min\left\{\left\lceil \frac{L_{in}^i}{k} \right\rceil\right\}$ and the minimum bandwidth to transmit kernel weights from external DRAM is calculated in the following equation:

$$BW = \frac{k^2 P^2 Q f}{min\left\{\left\lceil \frac{L_{in}^i}{k} \right\rceil\right\}}. \tag{12}$$

View Source

For pooling layer, the number of computation clocks are much less than that of convolutional layers. In the 2×2 max pooling pattern, each input feature map is cut down to a quarter of its original size. The computation clocks of $\frac{T_r}{2}$ rows in pooling layer j are calculated as follows:

$$N_{cycles}^{pooing_j} = \left\lceil \frac{L_{in}^j}{k} \right\rceil \times \left\lceil \frac{n_{in}^j}{P} \right\rceil, \tag{13}$$

View Source

where P denotes the degree of parallelism, k is the kernel size, L_{in}^j represents the length of input feature map, and n_{in}^j is the number of feature maps of the input layer. We divide L_{in}^j by k because every k pixels in a row are packaged and stored, so k pixels are read out every cycle.

Moreover, the computation cycles of a whole layer are:

$$N_{cycles\_total}^{pooing_j} = N_{cycles}^{pooing_j} \times \left\lceil \frac{2W_{in}^j}{T_r} \right\rceil, \quad j = 3, 6, \ldots, \tag{14}$$

View Source

where T_r denotes the row-tiling factor and W_{in}^j represents the width of input feature map, respectively.

Comparing Eq. (10) with Eq. (13), their first two terms look similar. However, in VGG16 model, the product of the first two terms remains equal and the last term of Eq. (10) $\left\lceil \frac{n_{out}^i}{P} \right\rceil >> 1$, so $N_{cycles\_total}^{CONV_i} >> N_{cycles}^{pooing_j}$ and the pooling time can be saved if inter-layer parallel processing is applied.

For fully connected layer, the number of computation cycles is estimated in the following equations:

$$N_{cycles}^{FC} = \frac{L^l n_{in}^l n_{out}^l}{P^l}, \tag{15}$$

View Source

where L^l denotes the length of input feature map, n_{in}^l and n_{out}^l represent the channels of input layer and output layer, and P^l is the degree of parallelism of fully connected layer.

Similarly, the total computation cycles of a whole FC layer is shown in Eq. (16):

$$N_{cycles\_total}^{FC} = \frac{WL^l n_{in}^l n_{out}^l}{P^l}, \tag{16}$$

View Source

where W denotes the width of input layer.

The number of interval clocks between two partial outputs of FC layers is $\sum_i N_{cycles}^{CONV_i}$ (Note that $N_{cycles}^{CONV_i}$ can be repeated several times) and in general cases, N_{cycles}^{FC} is smaller than $\sum_i N_{cycles}^{CONV_i}$. Therefore, once the inter-layer parallel processing is utilized, the computation time of fully connected layer can be saved as well.

B. Storage and Energy Analysis

For embedded FPGA platforms, due to limited internal memory resources, it is alomst impossible to place a whole large DCNN model on chip. If intermediate pixel data transfer is totally avoided, with regular layer-wise implementation, the required memory storage is determined by the largest storage requirement of adjacent two layers. In VGG16 model, the largest storage requirement of a layer is the first convolutional layer and its next convolutional layer. Both of them contain 3.2M ($224 \times 224 \times 64$) feature map pixels and total 6.4M pixels need to be stored on chip.

With the proposed memory storage design, the data storage scheme is arranged in a novel way which divides a whole layer computation into a series of partial row computations. In VGG16 model, the total number of feature map pixels stored in on-chip memory is 463.5k. Our design reduces memory requirement by 14 times and pixel data is no longer required to be sent to off-chip DRAM. Because the write operation delay of DRAM is avoided, the efficiency of memory bandwidth is highly improved.

The energy intensive consumptions are in two aspects: 1) the multiplications in arithmetic computations; 2) the data exchanges between off-chip DRAM and internal memories.

In arithmetic aspect, the energy cost of a multiplication operation is 31 times that of an addition operation in a 45nm CMOS process [31]. In VGG16 model, the convolutional kernel size is universal 3×3 . The regular convolution between a 3×3 receptive field and a 3×3 kernel requires 9 multiplications and 8 additions to get an output pixel. With the proposed 3-parallel FCUs, it needs 18 (6×3) multiplications and 36 ($10 \times 3 + 3 \times 2$) additions to calculate 3 output pixels. In other words, to obtain one output, it only requires 6 multiplications and 12 additions. It is estimated that 33% of arithmetic computing energy can be saved approximately.

In memory aspect, DRAM access consumes two orders of magnitude more energy than SRAM and has much more latency. The traditional memory architecture needs frequent access to external memory and can not meet the low power requirement of embedded system. The performance of high throughput is restricted due to high latency as well.

On two occasions data needs to be transferred from DRAM to on-chip RAMs in our architecture. One is during the loading time when the pixel data of input pictures need to be transferred. The other is the refresh of WBs during $computing\ phases$. The massive raw input pictures and the huge amount of weight parameters are stored in the DRAM. Since feature map pixels are not sent to DRAM, the latency of data transmission between external and internal memories are avoided. The energy is reduced because intermediate feature map data is no longer needed to be stored in the DRAM.

SECTION VIII. Implementation Results and Comparison

A. Implementation Analysis

The very deep CNN model, VGG16, is implemented as an experimental case in our design. We only show the average performances because the layer-wise implementation is not applied. The performance of fully connected layers is not constrained by the memory bandwidth, which is at least not the main factor. But the degree of computing parallelism is still limited by the bandwidth.

The implementation results are shown in Table. V. It is the same VGG16 model but of different computation degrees on platforms of Xilinx Zynq ZC706 and Virtex VC707. We achieve average performances of 316.23 GOP/s and 1250.21 GOP/s and the frame rates are 8.78 and 33.80 FPS, respectively.

B. Comparison Analysis

In Table. IV, we compare our design to other significant works on FPGA platforms in recent years. As shown in Table. IV, we put two hardware implementations of the same CNN model but with different computing degrees on FPGA platforms of Zynq XC7Z045 and Virtex7 VX485t, respectively. We achieve the highest performance (GOP/s) and resource utilization efficiency (GOP/s/slice).

TABLE IV Comparison With Other Works

Table IV- Comparison With Other Works

TABLE V Implementation Results of VGG16 on Two FPGA Platforms

Table V- Implementation Results of VGG16 on Two FPGA Platforms

Compared to regular CNN architectures, the advantages of the proposed architectures are shown as follows:

1) The proposed FCUs are computationally energy efficient because the multiplications are reduced. Based on the synthesis results, it is shown that the power of a 16-bit multiplier in the Design Compiler is approximately 20 times that of a 16-bit adder. The power of an FCU is dominated by the multipliers. It is estimated that 33% of the computational energy in regular convolutions with 3×3 kernel can be saved by applying 3-parallel FCUs. Moreover, due to the reduction of multiplications, the degree of parallelism in computation can be improved on embedded systems.

2) A novel data storage and reuse scheme are proposed. Data transfer is significantly reduced and restricted to data reads of input pictures and weights. Since the write transfer is avoided, we achieve a high degree of parallelism and an outstanding performance in our design. A further on-chip memory reuse method is also developed, which further reduces on-chip memory requirement without any overhead.

3) The ENQ method is employed in quantization process and it makes considerable reduction of the stored bits of pixel data, which further saves on-chip storage and energy.

SECTION IX. Conclusion

In this paper, we focus on the efficient hardware designs for CNN implementations. Convolutions dominate the computation complexity. While the latency of data transfer between external and internal memories is the main bottleneck of performance improvement. The fast FIR algorithm (FFA) is introduced and Fast Convolution Units (FCUs) are developed based on FFAs. By employing the parallel FCUs in the convolutions of CNN models, the computation complexity and energy are cut down significantly. The data storage and reuse scheme approach are especially proposed for data arrangement, which avoid the intermediate pixel data transfer between on-chip and off-chip memories. Hence the intermediate data transfer is avoided and meanwhile the throughput is largely improved. The ENQ method is also introduced to further reduce power and memory requirement. The VGG16 model is implemented on both Xilinx Zynq ZC706 and Virtex VC707 platforms and we achieve a frame rate of 8.76 FPS and 33.80 FPS with average performances of 316.23 GOP/s and 1250.21 GOP/s, respectively.

Authors

Figures

References

Citations

Keywords

Metrics

TECHNICAL INTERESTS CONTACT & SUPPORT