A Framework for Generating High Throughput CNN Implementations on FPGAs

Hanqing Zeng
University of Southern California
Ming Hsieh Department of Electrical Engineering
zengh@usc.edu

Ren Chen
University of Southern California
Ming Hsieh Department of Electrical Engineering
renchen@usc.edu

Chi Zhang
University of Southern California
Department of Computer Science
zhan527@usc.edu

Viktor Prasanna
University of Southern California
Ming Hsieh Department of Electrical Engineering
prasanna@usc.edu

ABSTRACT

We propose a framework to generate highly efficient accelerators for inferencing on FPGAs. Our framework consists of multiple algorithmic optimizations for computation complexity and communication volume reduction, a mapping methodology for efficient resource utilization, and a tool for automatic `Verilog` generation. The algorithmic optimizations improve throughput of frequency domain convolution so as to satisfy a given set of hardware constraints. While the Overlap-and-Add (OaA) technique has been known, it performs "wasted" computation at the edges. We propose a novel Concatenate-and-Pad (CaP) technique, which improves OaA significantly by reducing the "wasted" computation on the padded pixels. The proposed CaP used in conjunction with OaA enables us to choose a fixed FFT size at design time, and achieve low computation complexity for layers with various image sizes and kernel window sizes. We also develop a novel frequency domain loop tiling technique to further boost the throughput by improving data reuse. Our mapping methodology optimizes the architecture for the target device by fast design space exploration. We quantitatively categorize FPGAs by capturing their DSP resources, on-chip memory size and external memory bandwidth into a device coefficient. We identify the optimal architectural parameters based on the tradeoff between computation and communication cost. Our framework includes a tool to automatically generate fully synthesizable `Verilog`. We demonstrate the framework by generating high throughput accelerators for state-of-the-art CNN models on Intel HARP heterogeneous platform. Using our framework, we achieve throughput of 780.6 *GOPS*, 669.1 *GOPS* and 552.1 *GOPS* for AlexNet, VGG16 and FCN-16s respectively. These correspond to 6.8× (AlexNet) and 4.9× (VGG16) improvement compared with the state-of-the-art implementations.

KEYWORDS

Convolutional Neural Networks; Algorithmic Optimization; Hardware Mapping; Software-Hardware Co-design; FPGA;

ACM Reference Format:
Hanqing Zeng, Ren Chen, Chi Zhang, and Viktor Prasanna. 2018. A Framework for Generating High Throughput CNN Implementations on FPGAs. In *FPGA'18: 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, February 25–27, 2018, Monterey, CA, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3174243.3174265

1 INTRODUCTION

Convolutional Neural Networks (CNNs) are one of the most influential innovations in machine learning and computer vision [9, 15, 16]. With proliferation of deep learning models, the complexity and diversity of state-of-the-art CNNs has increased significantly.

Several challenges exist in accelerating CNNs on FPGAs:

- *Computation complexity*: Convolution layers of CNNs perform computationally expensive operations.
- *Hardware efficiency*: Efficiently accelerating various convolution layers is hard, due to the large variation of CNN model parameters across layers. The problems to be addressed are:
  - *Reconfiguration*: Hardware runtime reconfiguration can potentially meet the diverse computational requirements of various layers. However, time and resource overhead are incurred to support the flexibility in hardware.
  - *Wasted computation*: Using fixed hardware for acceleration avoids reconfiguration overhead. However, significant amount of computation can be wasted due to padding.
  - *Data reuse*: Given an on-chip memory of limited size, the accelerator needs to efficiently reuse on-chip data so as to reduce the communication volume to external memory.

Motivated by the above challenges, we propose a framework to generate high throughput accelerators for diverse CNN models. The inputs of the framework are the CNN model parameters (image size, kernel filter window size, number of input and output feature maps) and the FPGA device meta data (DSP resources, on-chip memory size and external bandwidth). The output is the automatically generated architecture on the target device specified in `Verilog`. To address the *computation complexity challenge*, our framework alleviates the computation burden of spatial convolution by frequency domain convolution. To address the *hardware utilization challenge*,

we solve the problems in *reconfiguration*, *wasted computation* and *data reuse* by multiple algorithmic optimizations. The Overlap-and-Add (OaA) operation has been used in [21] to implement frequency domain convolution with a fixed-size FFT module. We significantly improve the OaA approach by a novel Concatenate-and-Pad (CaP) operation. Compared with OaA, CaP achieves much lower computation complexity by filling the paddings with pixels from other images within the same batch. By applying CaP in conjunction with OaA, we identify a fixed FFT size in design time. Thus, without runtime reconfiguration, the accelerator achieves high throughput for layers with diverse image sizes and kernel filter window sizes. We further propose the frequency domain loop tiling technique which partitions the data blocks returned from the CaP-OaA step. Total communication volume to external memory is reduced as a result of increased reuse of on-chip data. In summary, our algorithmic optimizations perform light weight data layout rearrangement so that data from various layers are efficiently blocked into identical shapes before loaded onto FPGA. To optimize the architecture based on the FPGA resources, we then propose a hardware mapping methodology. A simplified performance model leads to fast design space exploration, which identifies the optimal architectural parameters by saturating the computation or communication capacity of the target device. Finally, a tool is developed to automatically generate implementations in `Verilog`. Our main contributions are:

- We propose algorithmic optimizations to improve throughput:
  - Concatenate-and-Pad (CaP) operation, a dual of Overlap-and-Add (OaA), which significantly improves the OaA based approach by reducing computation on paddings.
  - A data blocking methodology, which enables a fixed size FFT module to achieve low computation complexity for layers with various image and kernel window sizes.
  - Frequency domain loop tiling, which increases reuse of on-chip data by partitioning the feature map dimensions.
- We propose hardware mapping that incorporates the above algorithmic optimizations:
  - A generic architecture, which accelerates diverse CNNs on the target device without runtime reconfiguration.
  - A device coefficient, which measures the computation and communication capacity of the target FPGA by on-chip DSP resources, memory size and external bandwidth.
  - Fast design space exploration methodology, which identifies optimal architectural parameters on the target device.
- We develop a code generation tool that outputs fully synthesizable `Verilog` based on the resulting hardware mapping.
- We show that on Intel HARP platform, our techniques lead to throughput of 780.6 *GOPS*, 669.1 *GOPS* and 552.1 *GOPS* for AlexNet, VGG16 and FCN-16s respectively. The throughput corresponds to 6.8× (AlexNet) and 4.9× (VGG16) improvements compared with the state-of-the-art designs.

2 BACKGROUND

2.1 Frequency Domain Convolution and Overlap-and-Add (OaA)

We start from reviewing the convolution algorithm for 2D matrices.

Let I (shape: $l_{img} \times l_{img}$) and K (shape: $l_{kern} \times l_{kern}$) be the input and kernel matrices. Let M (shape: $l'_{img} \times l'_{img}$) be the output

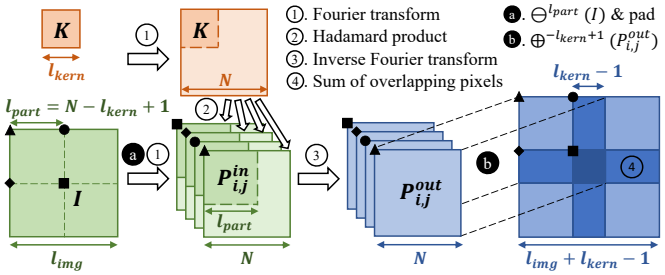


Figure 1: Frequency domain convolution using OaA

matrix. After transforming I and K into frequency domain representation, the sliding window operation of space convolution turns into the Hadamard product operation ($\circ$). *Equation 1* summarizes the algorithm for frequency domain convolution, where $\mathcal{F}$ and $\mathcal{F}^{-1}$ denote Fourier transform and its inverse operation.

$$M = I * K = \mathcal{F}^{-1}\Big(\mathcal{F}(I) \circ \mathcal{F}(K)\Big) \tag{1}$$

To perform the Hadamard product operation, dimensions l_{img}, l_{kern} of I, K need to be zero-padded to the same size before Fourier transform. When l_{img} is large (as is often the case for the first few convolution layers of a CNN), computing FFT on the complete I is not efficient. The Overlap-and-Add (OaA) technique [5] addresses the problem by partitioning I before the Fourier transform step.

The following describes the procedures of computing $I * K$ using OaA. Suppose we convolve I with K using N-point 2D FFT units (where $N > l_{kern} - 1$). First, we partition I into $P^{in}_{i,j}$ of shape $l_{part} \times l_{part}$ (where $l_{part} + l_{kern} - 1 = N$). Then, after zero padding $P^{in}_{i,j}$ to shape $N \times N$, we compute the intermediate output matrices $P^{out}_{i,j}$ using *Equation 2*. The final output matrix M is obtained by placing $P^{out}_{i,j}$ so that their up-left corners (pixel $(0,0)$) are located at pixels $(i \cdot l_{part}, j \cdot l_{part})$ of M. Value of each pixel in M is the sum of the overlapping pixels in $P^{out}_{i,j}$, as shown in *Equation 3* and *Figure 1*.

$$P^{out}_{i,j} = \mathcal{F}^{-1}\Big(\mathcal{F}(P^{in}_{i,j}) \circ \mathcal{F}(K)\Big) \tag{2}$$

$$M[p][q] = \sum_{i,j} \Big(P^{out}_{i,j}[p - i \cdot l_{part}][q - j \cdot l_{part}]\Big) \quad \text{where} \quad \begin{cases} 0 \leqslant p - i \cdot l_{part} < l_{part} \\ 0 \leqslant q - j \cdot l_{part} < l_{part} \end{cases} \tag{3}$$

Value enclosed by square brackets ($[*][*]$) indicates the pixel index within the matrix. All indices i, j, p, q start from 0.

We define operators $\ominus$ and $\oplus$. Operation $\ominus^{y}(I)$ partitions I into matrices of shape $y \times y$; $\oplus^{-x}(P)$ generates a large matrix from a set of matrices $\{P\}$ with x pixels overlapped, based on *Equation 3*.

2.2 Convolution Layers Using OaA

A convolution layer operates on a set of I and K, and outputs a set of M. Define I^{layer}, K^{layer} and M^{layer} as the high dimensional arrays of input, kernel filters and output feature maps of a layer. For batch processing, I^{layer}, K^{layer} and M^{layer} are of dimension $Batch \times f_{in} \times {l_{img}}^2$, $f_{out} \times f_{in} \times {l_{kern}}^2$ and $Batch \times f_{out} \times {l'_{img}}^2$ respectively, where f_{in}, f_{out} are number of input, output feature

maps. Let b, n and m index into the *Batch*, f_{in} and f_{out} dimensions. *Equation 4* specifies the operations of a convolution layer.

$$M_{b,m}^{layer} = \sum_{n<f_{in}} (I_{b,n}^{layer} * K_{m,n}^{layer}) \tag{4}$$

Algorithm 1 shows the operations of a convolution layer using OaA. Since K^{layer} is fixed for a trained CNN, we calculate $K^{freq} = \mathcal{F}(K^{layer})$ prior to the CNN inferencing computation.

Algorithm 1: Batch processing of a convolution layer using the OaA technique

Input: I^{layer} of shape $Batch \times f_{in} \times {l_{img}}^2$
K^{freq} of shape $f_{out} \times f_{in} \times N^2$
Output: M^{layer} of shape $Batch \times f_{out} \times {l'_{img}}^2$

for b = 0 to $(Batch - 1)$ **do**
  for i, j iterating matrices of $\ominus^{l_{part}}(I_b^{layer})$ **do**
    for n = 0 to $(f_{in} - 1)$ **do**
      $P_{n,i,j}^{in,freq} \leftarrow \mathcal{F}(P_{n,i,j}^{in,padded})$
    for m = 0 to $(f_{out} - 1)$ **do**
      for n = 0 to $(f_{in} - 1)$ **do**
        `// Element-wise MAC operation`
        $P' \leftarrow P_{n,i,j}^{in,freq} \circ K_{m,n}^{freq}$
        $P_{m,i,j}^{out,freq} \leftarrow P_{m,i,j}^{out,freq} + P'$
      $P_{m,i,j}^{out} \leftarrow \mathcal{F}^{-1}(P_{m,i,j}^{out,freq})$
  for m = 0 to $(f_{out} - 1)$ **do**
    $M_{b,m}^{layer} \leftarrow \oplus^{-l_{kern}+1}(P_{m,*,*}^{out})$
return M^{layer}

2.3 CNN Applications and Models

Feature extraction is fundamental to many applications. With little preprocessing on input images, CNNs extract high dimensional features associated with receptive fields of various sizes. Thus, variations of CNNs can be developed for specific applications.

We select three large scale state-of-the-art CNNs: AlexNet [9], VGG16 [15] and FCN-16s [11]. AlexNet and VGG16 can perform the tasks of feature extraction as well as image classification. FCN-16s is designed specifically for image segmentation. For feature extraction of AlexNet and VGG16, we execute all the convolution, ReLU and pooling layers, and skip the final fully connected layers. The input images can be of any l_{img} value. For image classification of AlexNet and VGG16, we execute all the layers including the fully-connected layers. The input images are scaled to be 224×224 pixels before feeding into the networks. For semantic segmentation of FCN-16s, we deploy deconvolution layers to replace fully connected layers. FCN-16s takes images of any l_{img} value as its input.

The above three CNNs are representatives of a wide range of recently developed deep CNNs. In general, the model parameters l_{img}, l_{kern}, f_{in} and f_{out} change dramatically from the first convolution layer to the last. As an example, *Table 1* summarizes the variation of these parameters for AlexNet, VGG16 and FCN-16s.

Table 1: Variation of model parameters

CNN	Conv. Layers	l_{kern}	$\frac{\max l_{img}}{\min l_{img}}$	$\frac{\max f_{in}}{\min f_{in}}$	$\frac{\max f_{out}}{\min f_{out}}$
AlexNet	5	11,5,3	17	128	4
VGG16	13	3	16	170	8
FCN-16s	18	7,3,1	50	1365	64

3 ALGORITHMIC OPTIMIZATIONS

Due to the large variation of l_{kern}, l_{img}, f_{in} and f_{out}, using a fixed architecture to accelerate various CNN models, or even for various layers of the same CNN model is very challenging. We show in this section three algorithmic optimizations to block input data into identical shapes after data layout rearrangement. As a result, computation complexity and communication cost are reduced, and a fixed hardware architecture (*Section 4*) on a target FPGA efficiently accelerates various CNNs. We show the procedure of hardware mapping and performance analysis in *Section 5*.

3.1 OaA Using Fixed FFT Size

For *native* frequency domain convolution, the FFT size is equal to $(l_{img} + l_{kern} - 1)$ and FFT is applied to the complete I at once without partitioning. The native approach is hard to be realized by accelerators, as hardware does not efficiently support FFT of arbitrary sizes. Previous work [21] addressed the hardware limitation by using the OaA technique. Their complexity analysis was based on 2D convolution without considering the f_{in} and f_{out} dimensions. We show in this section a more accurate complexity analysis on high dimensional convolution performed by a CNN layer. We also discuss how to select an appropriate FFT size for various l_{kern}.

According to *Algorithm 1*, for an FFT size N, number of operations performed by a convolution layer is calculated as:

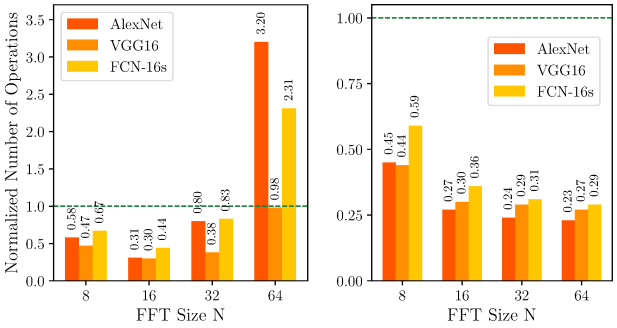
$$O_{total} = \Big(O_{part,FFT} + O_{part,MAC} + O_{part,IFFT} + O_{part,OaA}\Big) \cdot \Big\lceil \frac{l_{img}}{N - l_{kern} + 1} \Big\rceil^2 \tag{5}$$

where:

$$O_{part,FFT} = C_1 \cdot N^2 \cdot \log N \cdot f_{in}$$
$$O_{part,IFFT} = C_1 \cdot N^2 \cdot \log N \cdot f_{out}$$
$$O_{part,MAC} = C_2 \cdot N^2 \cdot f_{in} \cdot f_{out}$$
$$O_{part,OaA} = C_3 \cdot N \cdot (l_{kern} - 1) \cdot f_{out}$$

C_* are constants reflecting the cost of addition or multiplication.

We perform the following approximation to *Equation 5*: (1) Ignoring $O_{part,FFT}$ and $O_{part,IFFT}$: OaA performs partitioning of matrix I (l_{img}: order of 10^1 or 10^2), so it is reasonable to assume N to be in the order of 10^1. Observing that f_{in} and f_{out} are typically in the order of 10^2, the coefficient $\log N$ of $O_{part,FFT}$ ($O_{part,IFFT}$) is negligible compared with the coefficient f_{out} (f_{in}) of $O_{part,MAC}$. (2) Ignoring $O_{part,OaA}$: As required by OaA, $(l_{kern} - 1)$ should be less than N. So $O_{part,OaA} < C_3 \cdot N^2 \cdot f_{out} \ll O_{part,MAC}$. (3) Ignoring the ceiling function: We will discuss the effect of $\lceil * \rceil$ in detail in *Section 3.2*. We approximate *Equation 5* by:



(a) OaA Approach (b) CaP-OaA Approach

Figure 2: Number of operations for three CNNs. For AlexNet, we exclude the first convolution layer, as 8-point FFT cannot be applied to l_{kern} of 11 using OaA. The images input to the CNNs are of size 224×224, 224×224 and 500×500 for AlexNet, VGG16 and FCN-16s respectively.

$$O_{total}^{approx} \approx C_2 \cdot f_{in} \cdot f_{out} \cdot \left(\frac{l_{img}}{1-(l_{kern}-1)/N}\right)^2 \quad (6)$$

We conclude that:

(1) Number of operations decreases as N increases;
(2) Benefit of increasing N diminishes when N is sufficiently larger than $(l_{kern}-1)$.

From *Table 1*, we observe that for the three CNNs, all l_{kern} are less than 10 (except the first convolution layer of AlexNet). Taking the radix-2n FFT architecture [6] as an example, this means that setting N to be as small as 16 likely results in low enough computation complexity. *Figure 2a* shows the number of operations for AlexNet, VGG16 and FCN-16s according to *Equation 5*. Number of operations for OaA using various N (bars) are normalized by number of operations for spatial convolution (dashed line).

We observe that the best configuration is $N = 16$. If N keeps increases, computation complexity increases as opposed to the conclusion on *Equation 6*. This is because when N is 32 or 64, value of N is much larger than l_{img} of deep layers. Ceiling function in *Equation 5* then comes into picture. We show in *Section 3.2* the CaP operation to address this issue and justify our approximation to the ceiling function. We also observe that by applying OaA with a 16-point FFT module, we achieve significant reduction compared with spatial convolution. OaA using an uniform FFT size thus potentially processes convolution layers of various l_{kern} very efficiently. Note that our computation complexity is even much lower than the results in [21]. This is due to an additional optimization to utilize the imaginary channel for complex number operations (*Section 3.4*).

3.2 CaP for Reducing Wasted Computations

OaA requires the shape of each partition to be $N\times N$. The analysis on *Equation 6* ignores the useless computation on the zero paddings of $P_{i,j}^{in}$. Such approximation is not always valid, as can be seen in *Figure 2a* when $N = 32$ or 64. Two examples are shown in *Figure 3*. Scenario 1 is for deep layers when l_{img} is small and scenario 2 happens when l_{img} is larger.

One possible solution is to select an appropriate N which fits well l_{img} of most layers. The first problem is, this technique significantly

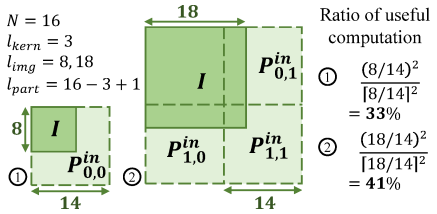


Figure 3: Examples showing that the majority of computation is performed on the padded pixels for OaA using $N = 16$

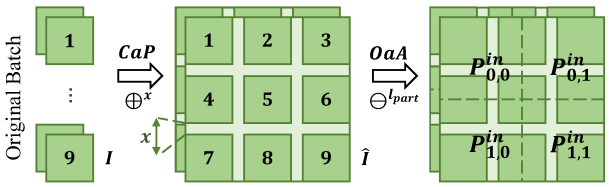


Figure 4: An example showing the CaP technique

limits the choices of N, leaving little freedom for architectural tuning on a target FPGA (*Section 5*). More importantly, identifying such an N value is often impossible. l_{img} can be of arbitrary value. This is especially the case for feature extraction applications, where images are not scaled to a fixed size before fed into the CNNs. An alternative solution is to mask the padded pixels. This saves number of operations, but still leads to low resource efficiency in hardware.

Instead of avoiding computation on paddings, we solve the problem from another perspective by filling the padded pixels with useful information. Based on *Equation 4*, out of the two dimensions involved in I^{layer} (*Batch* and f_{in}), f_{in} is shared between I^{layer} and K^{layer}. *Batch* is independent of K^{layer}. Thus, we fold the *Batch* dimension and expand I to solve the padding issue of OaA.

We call our operation Concatenate-and-Pad (CaP) [19]. Given a batch of d^2 images I of equal size $l_{img}\times l_{img}$, we arrange the images I in a $d\times d$ mesh, with x pixels of zero paddings between the vertically or horizontally adjacent images. CaP outputs a large image $\widehat{I}$ by concatenating multiple input images I. Parameter x is defined as the padding size of CaP. Parameter d is defined as the *Batch* folding factor. *Figure 4* illustrates how CaP reduces the wasted computation of OaA.

We observe the following with respect to the CaP operation. (1) *Aliasing* among adjacent images: The OaA operation that follows CaP may apply kernel windows covering pixels of multiple images (Step 2 in *Figure 1* and step OaA in *Figure 4*). It can be shown that aliasing among adjacent I in $\widehat{I}$ can be avoided iff $x \geqslant l_{kern}-1$. (2) *D*uality of the OaA and CaP operations: OaA *partitions* images, and CaP *combines* images. OaA processes a set of matrices by *overlapping* pixels (Step 4 in *Figure 1*), and CaP processes a set of matrices by *padding* pixels (Step CaP in *Figure 4*). Since CaP is a dual of OaA, we can extend the $\oplus^x$ operator (*Section 2.1*). If the superscript x is negative, then we use $\oplus^x$ to compute step b in *Figure 1*. If x is positive, we use $\oplus^x$ to compute $\widehat{I} = \oplus^x(I)$ in CaP.

In summary, we CaP the I_{layer} array so that input of $Batch\times f_{in}\times l_{img}{}^2$ is reshaped to $\frac{Batch}{d^2}\times f_{in}\times\widehat{l_{img}}^2$, where $\widehat{l_{img}} = d\cdot l_{img}+(d-1)\cdot(l_{kern}-1)$. We then apply OaA to $\widehat{I}$. Abbreviate such operations

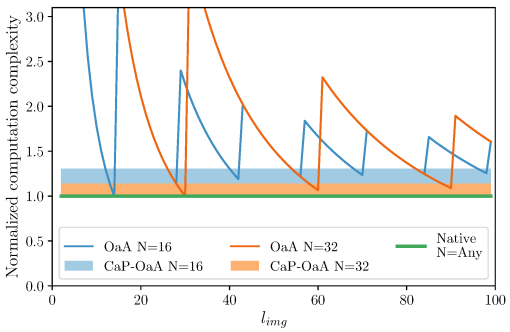


Figure 5: Comparison of computation complexity

as CaP-OaA. It is worth noticing that the various frequency domain convolution algorithms discussed so far are closely related to each other. CaP-OaA reduces to OaA when $d = 1$. OaA further reduces to native frequency domain convolution when $N \geqslant l_{img} + l_{kern} - 1$. Therefore, CaP-OaA is the most general version among these frequency domain convolution algorithms. CaP-OaA also achieves the highest hardware efficiency.

We further quantitatively analyze the computation complexity of CaP-OaA. CaP introduces a new variable d whose value can be set to approximate the ceiling function in *Equation 5*. It can be shown that by setting $d = \frac{N-(l_{kern}-1)}{\gcd(l_{img}+l_{kern}-1, N-(l_{kern}-1))}$ (where gcd means Greatest Common Divisor), the complexity of CaP-OaA is:

$$O_{CaP-OaA} < C_2 \cdot f_{in} \cdot f_{out} \cdot \left(\frac{l_{img} + l_{kern} - 1}{1 - (l_{kern} - 1)/N}\right)^2 \tag{7}$$

We compare OaA and CaP-OaA with the native approach as follow, where $O_{native} = C_2 \cdot f_{in} \cdot f_{out} \cdot (l_{img} + l_{kern} - 1)^2$.

$$\begin{aligned} \frac{O_{OaA}}{O_{native}} &= \left(\frac{\left\lceil \frac{l_{img}}{N-l_{kern}+1} \right\rceil \cdot N}{l_{img} + l_{kern} - 1}\right)^2 \\ \frac{O_{CaP-OaA}}{O_{native}} &< \left(\frac{1}{1 - (l_{kern} - 1)/N}\right)^2 \end{aligned} \tag{8}$$

The native approach achieves the lowest computation complexity compared with OaA and CaP-OaA. However, it requires the hardware to support FFT of arbitrary size. In the best case, OaA requires the same amount of computation as the native approach. Yet performance of OaA is highly dependent on N. As for CaP-OaA, as long as N is sufficiently larger than $(l_{kern} - 1)$, it ensures its computation complexity to be close to the native approach.

Figure 5 verifies our computation complexity analysis on the three algorithms. We vary l_{img} from 3 to 100, and fix l_{kern} to be 3. Complexity of CaP-OaA and OaA are normalized by complexity of the native approach. Colored areas for CaP-OaA show the possible ranges of its computation complexity according to *Equation 8*.

Figure 2b shows the number of operations for three CNNs using CaP-OaA. Compared with *Figure 2a*, we conclude that given any fixed N, CaP-OaA achieves low computation complexity consistently for convolution layers of various l_{kern} and arbitrary l_{img}.

3.3 Frequency Domain Loop Tiling

The CaP-OaA technique manipulates the data dimensions l_{img} and l_{kern}. To block data of convolution layers into identical shapes, we still need optimization on the f_{in} and f_{out} dimensions.

We revisit *Algorithm 1*. Tiling of the loop dimensions in lines 5 and 6 performs partitioning of f_{in} and f_{out}. In runtime, the kernel filters and image data are partitioned into fixed shapes, and the tiles are loaded onto FPGA. Tiling on top of CaP-OaA makes the data flow of diverse CNNs on a target device identical to each other. The tiling factor f is the same for various convolution layers. After CaP-OaA transforms the kernel filters and images to an uniform $N \times N$ shape, value of f becomes independent of the CNN model parameters, and is solely bound by the on-chip memory size. The motivation for loop tiling is to reduce the communication volume to external memory by increased reuse of on-chip data [4]. For frequency domain convolution, tradeoff exists between N and f to balance computation complexity and data reuse. Analysis on the algorithm-architecture co-design is made in *Section 5*.

Although loop optimization for CNNs on FPGAs has been extensively studied, previous work [4, 8, 12] focused on convolution in space domain. Existing techniques cannot be directly applied to frequency domain CNNs, since data flow of sliding window operations is different from Hadamard product operations. On the other hand, our three techniques proposed in *Section 3.1*, 3.2 and 3.3 can all be understood as loop optimizations in frequency domain. OaA is analogous to loop tiling of l_{img}, and CaP is analogous to loop tiling and unrolling of the *Batch* dimension.

With the optimizations in *Section 3.1*, 3.2 and 3.3, we derive *Algorithm 2* from *Algorithm 1*. Lines 6 to 13 shows the workload on FPGA. The rest of the algorithm specifies the operations by CPU. Loop unrolling of lines 9 and 10 is discussed in *Section 4*.

3.4 Composing a Complex Image

Fourier transform converts a real number image into complex number representation. A straightforward implementation feeds input image data to the real channel and zeros to the imaginary channel. To better utilize the hardware resources, a better implementation feeds two images within a batch to the real and imaginary channels simultaneously. Thus, given two images I_1, I_2 and kernel filter K, we perform $(I_1 + j \cdot I_2) * K = \mathcal{F}^{-1}\left(\mathcal{F}(I_1 + j \cdot I_2) \circ \mathcal{F}(K)\right) = (I_1 * K) + j \cdot (I_2 * K)$, where I_1, I_2 and K are all of real values.

Composing a complex image reduces the computation complexity *by half*, and *doubles* the efficiency of hardware DSPs. The technique in this section can be easily combined with OaA-CaP.

4 SYSTEM ARCHITECTURE

We define throughput (bandwidth) as the number of complex words transferred per unit time. Also, the number of DSPs used in implementing a complex multiplier-accumulator as the unit of DSP resources; bytes per complex word as the unit of on-chip memory. Hardware parallelism is measured in terms of number of parallel operations on complex data.

4.1 Data Reuse Scheme

Image or kernel oriented data reuse schemes have both been explored in previous work [3] for spatial convolution. In our design,

Algorithm 2: Batch processing of a convolution layer using CaP-OaA and f_{in}, f_{out} loop tiling

Input: I^{layer} of shape $Batch \times f_{in} \times {l_{img}}^2$
K^{freq} of shape $f_{out} \times f_{in} \times N^2$; $K^{freq} = \boldsymbol{\mathcal{F}}(K^{layer})$
Output: M^{layer} of shape $Batch \times f_{out} \times {l'_{img}}^2$

1 **for** $b = 0$ to $(Batch - 1)$, **stride by** D **do** // $D = d^2$
2 **for** $m = 0$ to $(f_{out} - 1)$, **stride by** f **do**
3 **for** $n = 0$ to $(f_{in} - 1)$, **stride by** f **do**
4 $\widehat{I^{tile}} \leftarrow \oplus^{l_{kern}-1}(I^{layer}_{b:b+D,n:n+f})$
5 $K^{tile,freq} \leftarrow K^{freq}_{m:m+f,n:n+f}$
 /* FPGA starts to process tiled data. */
6 **for** i, j iterating matrices of $\ominus^{l_{part}}(\widehat{I^{tile}})$ **do**
7 **for** $n' = 0$ to $(f - 1)$ **do**
8 $P^{in,freq}_{n',i,j} \leftarrow \boldsymbol{\mathcal{F}}(P^{in,padded}_{n',i,j})$
9 **for** $m' = 0$ to $(f - 1)$ **do**
10 **for** $n' = 0$ to $(f - 1)$ **do**
11 $P' \leftarrow P^{in,freq}_{n',i,j} \circ K^{tile,freq}_{m',n'}$
12 $P^{out,freq}_{m',i,j} \leftarrow P^{out,freq}_{m',i,j} + P'$
13 $P^{out}_{m',i,j} \leftarrow \boldsymbol{\mathcal{F}}^{-1}(P^{out,freq}_{m',i,j})$
 /* FPGA ends processing. */
14 **for** $m' = 0$ to $(f - 1)$ **do**
15 $M' \leftarrow \oplus^{-l_{kern}+1}(P^{out}_{m',*,*})$
16 $M'' \leftarrow$ Reshape M' to $D \times 1 \times (l_{img})^2$
17 $M^{layer}_{b:b+D,m+m'} \leftarrow M^{layer}_{b:b+D,m+m'} + M''$
18 **return** M^{layer}

the element-wise Hadamard product results in a clean data movement pattern, so we simply calculate the amount of data reuse to make our design choice. Reuse of image pixels are proportional to f, and reuse of kernel pixels are proportional to the batch size. We use the kernel-oriented data reuse scheme. Before execution, a tile of kernel filters is pre-loaded onto FPGA. The kernel loading time is amortized for a large enough batch. Our reuse scheme is equivalent to loop interchanging of line 1 with 2, 3 in *Algorithm 2*.

4.2 Overall System Design

Based on *Algorithm 2*, we design the hardware modules on FPGA to execute the workload from line 6 to 13. Prior to FPGA execution, a kernel filter tile $K^{tile,freq}$ is pre-loaded to on-chip memory BUF_K. When data streams in from external memory, a 2D FFT module transforms partitions of I^{layer} into frequency domain (lines 7, 8). Outputs of the FFT module are stored in the on-chip memory BUF_I. After reading matrices from BUF_I and BUF_K, the Hadamard-Accumulation (HAC) module performs element-wise multiplication-accumulation (lines 9 to 11). HAC feeds its accumulated outputs to a 2D IFFT module, which transforms the partitions back to space domain (line 13). The IFFT module sends its outputs directly to external memory. *Figure 6* shows the overall system design. Note

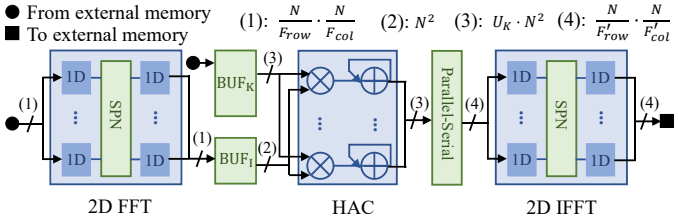


Figure 6: Overall FPGA architecture

that a small buffer is placed between the HAC and IFFT module. It serves as a parallel-serial converter. We will see later on that data parallelism of HAC should be larger than the IFFT module.

HAC module. The key benefit of frequency domain convolution is that sliding window operation in spatial convolution turns into Hadamard product operation. Thus, all loop carried dependencies are automatically eliminated. Massive parallelism can then be exploited by the HAC module. Based on the FPGA resources, we unroll the loop of line 9, *Algorithm 2*. Each cycle, HAC takes as input U_K slices of $N \times N$ matrices from BUF_K, and 1 slice from BUF_I. Each slice is also fully unrolled to a 1D array of length N^2. Thus, data parallelism of HAC is $U_K \cdot N^2$. Every f^2/U_K cycles, HAC traverses the f^2 slices of $K^{tile,freq}$. During the period, f slices of inputs are read from BUF_I, and each slice is reused for f times. On the output side, f slices are generated and fed in the IFFT module. Throughput of HAC is thus $U_K \cdot \frac{N^2}{f}$.

2D FFT (IFFT) module. FFT on a $N \times N$ complex number matrix involves two computation phases. In both phases, N-point 1D FFT is performed on each of the N rows of the 2D matrix. Input to phase 1 is the original matrix. Input to phase 2 is the transposed output matrix of phase 1. For each phase, a straightforward implementation deploys N 1D FFT pipelines where each 1D FFT pipeline supports data parallelism of N. Since the throughput of 2D FFT is bounded by the external memory bandwidth and the throughput of the HAC module, we may either increase the data parallelism of 2D FFT by unrolling the loop in line 7 in *Algorithm 2*, or decrease the parallelism by folding the FFT pipelines. Under the current memory technology, it is very unlikely that the external bandwidth is large enough to transfer more than N^2 complex words per cycle (N: order of 10^1 or 10^2). Thus, we set the FFT unrolling factor to be 1. Folding can be performed along each of the two dimensions. Let F_{col} and F_{row} be the column and row folding factors. Column folding reduces the number of 1D FFT pipelines from N to $\frac{N}{F_{col}}$. Row folding reduces data parallelism of each 1D FFT pipeline from N to $\frac{N}{F_{row}}$ [2]. Data parallelism of the 2D FFT module is thus $\frac{N}{F_{col}} \cdot \frac{N}{F_{row}}$.

Matrix transpose between phase 1 and phase 2 for the 2D FFT architecture is implemented by a Streaming Permutation Network (SPN) [2]. The resource efficient in-place permutation in time algorithm of SPN requires a single port memory of size N^2 to support data parallelism of $\frac{N}{F_{col}} \cdot \frac{N}{F_{row}}$.

For the 2D IFFT module, we use the same architectural parameters as 2D FFT (F_{col} and F_{row}).

On-chip memory. Data in BUF_I are reused f times (f/U_K times temporal reuse, U_K times spatial reuse) before they are replaced by the next tile. Thus, we use the double buffering technique for BUF_I.

Data communication latency is completely hidden if throughput of HAC is no less than the streaming FFT module (*Section 5.1*). BUF_K and BUF_I store one kernel tile and one image tile respectively. So the BUF_K size is $M_K = f^2 \cdot N^2$ and the BUF_I size is $M_I = f \cdot N^2$.

5 ARCHITECTURE MAPPING

Define **M**, **L** as the total memory size and DSP resources on chip, **B** as the total external memory bandwidth.

5.1 Performance Model

By our data reuse scheme, we ignore the kernel communication cost and utilize the full bandwidth **B** to read and write image tiles.

By calculating the throughput of each individual hardware module, we derive the overall system throughput R_{sys}:

$$R_{sys} = \min\left\{\frac{N}{F_{col}} \cdot \frac{N}{F_{row}}, \quad U_K \cdot \frac{N^2}{f}, \quad \frac{1}{2} \cdot \mathbf{B}\right\} \tag{9}$$

To execute one layer for a batch of complex images, we first keep one kernel tile in BUF_K, and load the image tiles belonging to $\widehat{I}$ one by one into BUF_I. Then we replace the data in BUF_K with the next kernel tile belonging to the same layer, and repeat the loading of image tiles. The execution time averaged for one input I is:

$$t_{img} = \left\lceil \frac{f_{in}}{f} \right\rceil \cdot \left\lceil \frac{f_{out}}{f} \right\rceil \cdot \left\lceil \frac{\widehat{l_{img}}}{N - l_{kern} + 1} \right\rceil^2 \times \frac{f \cdot N^2}{R_{sys}} \times \frac{1}{d^2} \tag{10}$$

We may further simplify R_{sys}. First of all, we observe that the architectural parameters should be set such that throughput of FFT, IFFT and HAC are matched. Secondly, for **L** and **M**, we observe that: (1) Most of the DSP resources are consumed by the HAC module to perform Hadamard product. This observation is consistent with the conclusion in *Section 3.1*. (2) Most of the on-chip memory is consumed by the kernel tile. The size of the kernel tile is in the order of $f^2 \cdot N^2$, and the size of the image tile is in the order of $f \cdot N^2$. Thus, we approximate R_{sys} as follows.

$$R_{sys} = \min\left\{N \cdot \frac{\mathbf{L}}{\sqrt{\mathbf{M}}}, \quad \frac{1}{2} \cdot \mathbf{B}\right\} \tag{11}$$

We then get throughput of a convolution layer by t_{img} and R_{sys}.

5.2 Device Coefficient

The two terms in *Equation 11* show the potential computation and communication bounds of the target device. As N increases, the system shifts from being computation bound to communication bound. We analyze the system performance for various N. Based on this, we quantitatively categorize FPGA devices by **L**, **M** and **B**.

Case 1: small N. The system throughput R_{sys} is determined by **L** and **M**. So $R_{sys} = N \cdot \frac{\mathbf{L}}{\sqrt{\mathbf{M}}}$. By approximating $\left\lceil \frac{f_{in}}{f} \right\rceil \cdot \left\lceil \frac{f_{out}}{f} \right\rceil \approx \frac{f_{in}}{f} \cdot \frac{f_{out}}{f}$ and $\left\lceil \frac{\widehat{l_{img}}}{N - l_{kern} + 1} \right\rceil \approx \frac{d \cdot (l_{img} + l_{kern} - 1)}{N - l_{kern} + 1}$. We update *Equation 10*:

$$t'_{img} \approx f_{in} \cdot f_{out} \cdot (l_{img} + l_{kern} - 1)^2 \cdot \left(\frac{N}{N - l_{kern} + 1}\right)^2 \cdot \frac{1}{\mathbf{L}} \tag{12}$$

Case 2: large N. The system throughput R_{sys} is determined by **B**. Therefore, $R_{sys} = \frac{1}{2} \cdot \mathbf{B}$. By approximating $\left\lceil \frac{f_{in}}{f} \right\rceil \cdot \left\lceil \frac{f_{out}}{f} \right\rceil \approx \frac{f_{in}}{f} \cdot \frac{f_{out}}{f}$ and $\frac{N}{N - l_{kern} + 1} \approx 1$. We derive *Equation 10* as:

$$t''_{img} \approx f_{in} \cdot f_{out} \cdot (l_{img} + l_{kern} - 1)^2 \cdot (2 \cdot N) \cdot \frac{1}{\sqrt{\mathbf{M}} \cdot \mathbf{B}} \tag{13}$$

As N grows from a small value, latency t'_{img} decreases due to lower computation complexity (*Section 3.1*). However, with limited on-chip memory size, larger N means smaller f and lesser data reuse. At some point, external bandwidth saturates. As N increases, latency t''_{img} becomes larger, since communication cost then becomes the dominant factor. When sweeping N, expressions on the right side of *Equation 12* and *13* forms two curves, which are the performance asymptotic bounds. We define the common term $f_{in} \cdot f_{out} \cdot (l_{img} + l_{kern} - 1)^2$ as the *model coefficient* $\mathbf{K}_\text{CNN}$, which scales performance of the architecture by the complexity of CNN.

Imagine a device with infinite bandwidth **B**. It is never bound by communication. Thus, its maximal achievable throughput according to *Equation 12* is $R^{max}_{comp} = \max \frac{1}{t'_{img}} = \frac{1}{\mathbf{K}_\text{CNN}} \cdot \mathbf{L}$, which is the theoretical computational upper bound.

Dividing the reciprocal of t'_{img} and t''_{img} by R^{max}_{comp}, we get the asymptotic bounds for normalized throughput:

$$\begin{aligned} C_{comp}(N) &= \left(\frac{N - l_{kern} + 1}{N}\right)^2 \\ C_{comm}(N) &= \left(\mathbf{B} \cdot \mathbf{M}^{\frac{1}{2}} \cdot \mathbf{L}^{-1}\right) \cdot \left(\frac{1}{2 \cdot N}\right) \end{aligned} \tag{14}$$

We define $\mathbf{K}_\text{FPGA} = \mathbf{B} \cdot \mathbf{M}^{\frac{1}{2}} \cdot \mathbf{L}^{-1}$ as the *device coefficient*. For a target device, it measures the ratio of communication capacity over computation capacity. Based on $\mathbf{K}_\text{FPGA}$, we map the architecture onto the target device by balancing the computation complexity and data reuse. Details of the mapping are shown in the next section.

5.3 Design Space Exploration

We note some important properties of C_{comp} and C_{comm}. The model coefficient $\mathbf{K}_\text{CNN}$ disappears in the process of normalization. Furthermore, $\mathbf{K}_\text{FPGA}$ captures the device characteristics in C_{comm}. For diverse CNNs, we can use a constant l_{kern} (= 3) to approximate C_{comp}, so the computation bound is a single curve. On the other hand, for a given target device, the communication bound is also a single curve since $\mathbf{K}_\text{FPGA}$ keeps as a constant regardless of the CNNs. In this sense, $\mathbf{K}_\text{FPGA}$ intrinsically determines the device performance, and the normalized throughput is independent of the CNN model parameters.

With $\mathbf{K}_\text{FPGA}$, design space exploration is as simple as identifying the intersection point of two curves. We propose a *design chart* (*Figure 7*). The red and blue solid lines are the computation and communication rooflines bounding the actual performance. Intersection of the rooflines shows the optimal N for the target device. Parameters f, U_K, F_{row} and F_{col} are calculated based on N. *Algorithm 3* shows the procedure for a radix-2n FFT architecture.

In the design chart, we use the device coefficient ($\mathbf{K}_\text{Stratix-V}$) of our experimental platform as reference. For this device, N = 16 is the best configuration. For devices with their coefficients

Algorithm 3: Design space exploration using the design chart

Input : L, M, B of the target device
Output: N chosen for the architecture

1 $\mathbf{K_{FPGA}} \leftarrow \mathbf{B} \cdot \mathbf{M}^{\frac{1}{2}} \cdot \mathbf{L}^{-1}$
2 Get the curve of communication bound $C_{comm}(N)$
3 $N_0 \leftarrow$Intersection of $C_{comm}(N)$ and $C_{comp}(N)$
4 $N' \leftarrow 2^{\lfloor \log_2 N_0 \rfloor}$; $N'' \leftarrow 2^{\lceil \log_2 N_0 \rceil}$
5 **return** $C_{comp}(N') > C_{comm}(N'')$? $N' : N''$

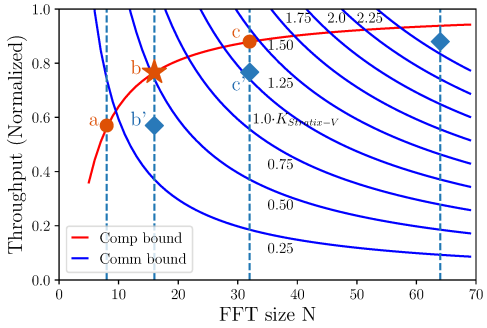


Figure 7: Design chart for hardware mapping

falling between 0.2 to 0.35, 0.52 to 1.05, 1.45 to 2.37, the designs are computation bound. Optimal N is 8, 16 and 32 respectively (these are shown by three red marks a, b, c). Similarly, for other devices, the designs are communication bound (design points falling between b and b', c and c'). Using the design chart, we can identify target devices that are best suited for our architecture. Devices with their roofline intersections falling at the blue vertical lines (N = 8, 16, 32, 64) have perfectly balanced resources in terms of **L**, **M** and **B** (e.g., devices with $\mathbf{K_{FPGA}}$ = 0.2, 0.5 or 1.18).

6 AUTOMATIC CODE GENERATION

We have developed a tool [20] to automatically generate the architecture on the target device. *Figure 8* shows the workflow of the tool. The inputs are the CNN model parameters for each convolution layer (l_{img}, l_{kern}, f_{in} and f_{out}), and the meta data of the target device (**B**, **L** and **M**). The outputs includes `C++` code for book-keeping the data blocks (lines 1-5 and 14-18, *Algorithm 2*), and synthesizable `Verilog` performing the computational expensive convolution (lines 6-13, *Algorithm 2*). The Mapping Engine feeds the CaP-OaA parameters (N, d) and tiling factor (f) into Software Generation Engine, and feeds architectural parameters into Hardware Generation Engine. Optionally, users can specify additional constraints to the tool such as available FFT sizes and maximum d.

Software Generation. Although the optimal batch folding factor d varies across convolution layers, we use a uniform d for all layers of a CNN in implementation. This ensures that the output of the previous layer can be directly fed into the following layer without further layout rearrangement.

Hardware Generation. The 2D FFT module consists of 1D FFT pipelines and Streaming Permutation Networks (SPN) for matrix

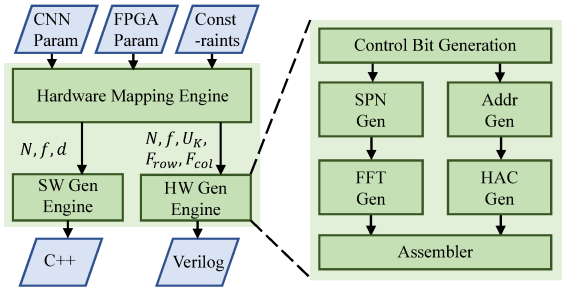


Figure 8: Tool workflow

transpose. We take the 1D FFT template from [13]. SPN is a folded CLOS network including two spatial permutation stages and one temporal permutation stage. We implement the in-place permutation in time algorithm [2] to generate the control bits. HAC includes a memory controller to fetch data from BUF_I and BUF_K. Since our architecture does not involve any runtime reconfiguration, Hardware Generation Engine statically computes all the SPN control bits and HAC input addresses in design time. The Assembler connects the 2D FFT, 2D IFFT, HAC, BUF_I and BUF_K based on *Figure 6*.

7 EXPERIMENTAL RESULTS

7.1 Experimental Setup

We use Intel Heterogeneous Research Platform (HARP) [1] for evaluation. HARP has shared memory accessible to the CPU and FPGA. The FPGA is an Intel Stratix V GXA7 device, with 5 *GB/s* bandwidth to external memory, 6.25 *MB* on-chip memory, 256 *DSPs* and 234720 *ALMs*. The CPU of HARP is a 10-core Intel Xeon E5-2600 v2 processor. We use 16-bit fixed-point data representation to compute CNNs. The designs were synthesized by Quartus II (version 13.1.0).

In the following, throughput is calculated as the total number of operations for *spatial* convolution divided by the average execution time per image for our *frequency domain* approach. Numerator for spatial convolution let us make fair comparison with other works. The execution time is the actual execution time on HARP.

The architecture for *all* CNNs under evaluation is configured as: N = 16, f = 64, $U_K = \frac{1}{2}$, F_{row} = 4, F_{col} = 16. We set an upper limit for d ($\leqslant$ 15) to bound the batch size.

For workload distribution between FPGA and CPU, FPGA executes all convolution layers of AlexNet, VGG16 and FCN-16s except the first convolution layer of AlexNet, while the CPU executes all the remaining layers (pooling, ReLU, fully connected and first convolution of AlexNet). In summary, the CPU executes 15%, 1% and 1% of the total computation for AlexNet, VGG16 and FCN-16s respectively. We implement the first convolution layer of AlexNet using the BLAS [17] library. By a simple batch processing pipeline, execution time of CPU is completely hidden by FPGA.

7.2 Impact of Algorithmic Optimizations

To vary the input image size, we use AlexNet and VGG16 to execute feature extraction by skipping their fully-connected layers. We execute all layers of FCN-16s.

Effect of CaP. We use the architecture configuration as specified in *Section 7.1*. We vary l_{img} of the first convolution layer from 160

to 304 for AlexNet and VGG16, and from 320 to 608 for FCN-16s (In other words, l'_{img} of the last convolution layer for the three CNNs vary from 10 to 19). *Figure 9* shows the comparison of computation complexity for frequency domain convolution using CaP-OaA, OaA and spatial convolution. Each bar is vertically stacked by the number of operations for each convolution layer of the CNNs. *Figure 10* shows the comparison of the measured throughput on HARP.

When l_{img} is divisible by $(N - l_{kern} + 1)$ (e.g., $l_{img} = 224$ for AlexNet and VGG16), performance of OaA is identical to CaP-OaA. However, in other cases, CaP-OaA delivers much better performance than OaA. For example, when $l_{img} = 240$ for AlexNet, VGG16 and $l_{img} = 352$ for FCN-16s, CaP-OaA leads to 2.3×, 1.5× and 1.7× complexity reduction, and 2.3×, 1.5× and 1.7× throughput improvement. Furthermore, we observe that the performance of OaA is highly sensitive to image sizes. For AlexNet and VGG16, performance drops significantly when the image size increases from 224 to 240. This reflects the padding effect of OaA.

Effect of N. Next, we experiment how selecting various N affects the throughput of the system. Since parameters N and f are together dependent on the on-chip memory size, by varying N, we are exploring the effect of loop tiling as well. *Figure 11* shows the normalized throughput of the three CNNs on the design chart when using $N = 8, 16, 32$. The corresponding f values are $128, 64, 32$.

As predicted by the design chart, $N = 16$ is the best configuration on the Stratix-V GXA7 device. When $N = 8$, the increased computation complexity degrades the performance. When $N = 32$, the low data reuse makes external bandwidth the bottleneck. Furthermore, despite the dramatically different network structure, the normalized throughput of the three CNNs are very close to each other. This demonstrates the effect of our algorithmic optimization.

7.3 Comparison with State-of-the-Art

For AlexNet and VGG16, we use the ImageNet dataset ($l_{img} = 224$).

Table 2 summarizes the comparison with state-of-the-art designs. All the designs except [21] use similar or lower precision data representation than our designs. In [21], frequency domain convolution using the OaA technique was employed. However, their analysis was based on a metric called "delay-multiplier product" evaluating convolution of a single image rather than a complete layer. Using the same FPGA, we show 9.4× (AlexNet) and 5.4× (VGG16) speed up in throughput as a result of a deeper analysis on frequency domain convolution. All other works are based on spatial convolution. Compared with [18] which uses the same target FPGA and data representation as this project, we achieve 5.8× speedup. Compared with [7], [8] and [12], when we use the same data representation (16-bit fixed point), our designs achieve 1.4×, 4.9× and 1.0× speedup, even though our target device has 14.0×, 3.4× and 5.9× less DSP resources. Using a device with 5.9× more DSPs, [22] achieves 2.7× higher throughput than us. One main reason is the difference in the clock rate. We can not achieve higher clock rate, since HARP requires the FPGA to operate at exactly 200 MHz.

To understand such significant improvement in throughput, we use [18] as an example to show the improvement breakdown. Out of the 5.8× improvement, approximately 3× comes from the reduction in computation complexity (*Figure 2b*). The remaining 2× comes from the clock rate improvement. The Hadamard product operation leads to much less number of operations and much simpler data flow compared with the sliding window operation.

To the best of our knowledge, this is the first work that accelerates FCN-16s on FPGAs. As shown in *Figure 10*, approximately, throughput of 550 *GOPS* is achieved for images of various sizes.

8 RELATED WORK

Accelerating spatial convolution has been extensively studied from the perspective of loop operation optimization [4, 12] and data flow optimization [3]. Work in [4] proposed a roofline model to capture various techniques including loop tiling, unrolling and interchanging. [12] further optimized performance by a thorough design space exploration. [22] boosted throughput under the OpenCL framework. Spatial convolution based approaches will eventually be bound by the computation complexity of the convolution algorithm. On the other hand, alternatives such as convolution by Winograd transform and frequency domain convolution have been proposed and implemented [10, 14, 21]. Winograd based approaches do not easily generalize to CNNs with various kernel window sizes. While the approaches based on frequency domain convolution are more flexible, further optimizations to [21] can be performed when processing high dimensional data of convolution layers (this work).

9 CONCLUSION

We presented a framework for generating high throughput CNN accelerators. Combining the CaP, OaA and frequency domain loop tiling techniques together, our framework generates architectures accelerating diverse CNNs without runtime reconfiguration.

In the future, we will explore the hybrid algorithm combining convolution in space and frequency domain. Spatial convolution is as efficient as frequency domain convolution for 1×1 kernels. In such cases, we may switch to spatial convolution which leads to better hardware utilization. In addition, as techniques have been developed to make use of the sparsity in spatial convolution, we will explore if similar techniques can be applied in frequency domain.

10 ACKNOWLEDGEMENTS

This work was supported by the US NSF under grants CNS-1643351, ACI-1339756 and CCF-1320211. This work is also supported in part by Intel Strategic Research Alliance funding. Equipment grant from the Intel Hardware Accelerator Research Program is gratefully acknowledged.

REFERENCES

[1] 2015. Intel Inc. Xeon+FPGA Platform for the Data Center. (2015). https://www.ece.cmu.edu/calcm/carl/lib/exe/fetch.php?media=carl15-gupta.pdf
[2] R. Chen, H. Le, and V. K. Prasanna. 2013. Energy efficient parameterized FFT architecture. In *2013 23rd Intl. Conf. on Field programmable Logic and Applications.*
[3] Y. H. Chen, J. Emer, and V. Sze. 2017. Using Dataflow to Optimize Energy Efficiency of Deep Neural Network Accelerators. *IEEE Micro* 37, 3 (2017).
[4] Chen Zhang, et al. 2015. Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '15).* ACM.
[5] Ali Daher, et al. 2010. Overlap-save and overlap-add filters: Optimal design and comparison. *IEEE Transactions on Signal Processing* 58, 6 (2010).
[6] P. Duhamel and H. Hollmann. 1984. 'Split radix' FFT algorithm. *Electronics Letters* 20, 1 (January 1984).
[7] Huimin Li, et al. 2016. A high performance FPGA-based accelerator for large-scale convolutional neural networks. In *2016 26th International Conference on Field Programmable Logic and Applications (FPL).*

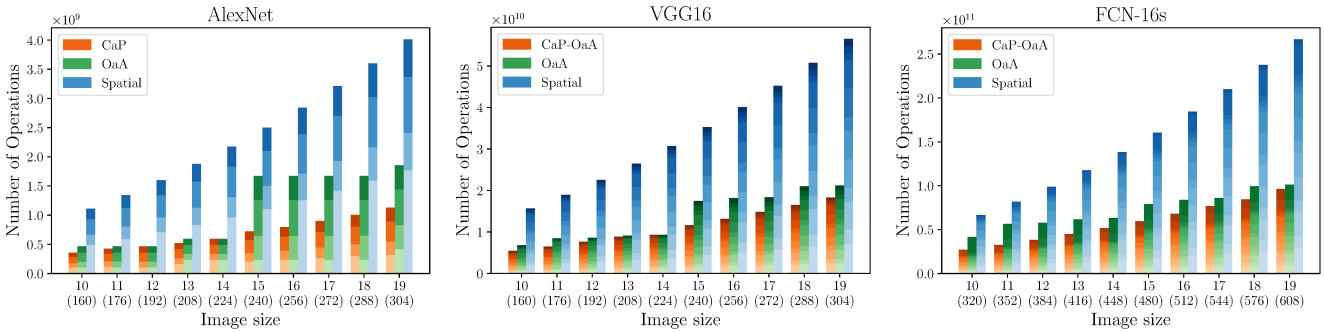


Figure 9: Number of operations performed by various convolution algorithms

Table 2: Comparison with state-of-the-art AlexNet and VGG16 implementations (FX: fixed point, FT: floating point)

	[7] AlexNet	[18] AlexNet	[21] AlexNet	[8] VGG16	[12] VGG16	[22] VGG16	[21] VGG16	Proposed: AlexNet	Proposed: VGG16
FPGA	Virtex-7 VC709	Stratix-V GXA7	Startix-V GXA7	Zync XC7Z045	Arria-10 GX1150	Arria-10 GX1150	Stratix-V GXA7	Stratix-V GXA7	Stratix-V GXA7
Frequency (*MHz*)	156	100	200	150	150	385	200	200	200
Precision	16 bit FX	8-16 bit FX	32 bit FT	16 bit FX	8-16 bit FX	16 bit FX	32 bit FT	16 bit FX	16 bit FX
DSP Usage	2144 (60%)	256 (100%)	224 (88%)	780 (89%)	1518 (100%)	1378 (91%)	224 (88%)	256 (100%)	256 (100%)
Logic Usage	274K (63%)	121K (52%)	200K (85%)	183K (84%)	161K (38%)	–	200K (85%)	107K (46%)	107K (46%)
On-chip RAM	956 (65%)	1152 (61%)	1208 (64%)	486 (87%)	1900 (70%)	1450 (53%)	1208 (64%)	1377 (73%)	1377 (73%)
Throughput (*GOPS*)	565.9	134.1	83.0	137.0	645.3	1790	123.5	780.6	669.1

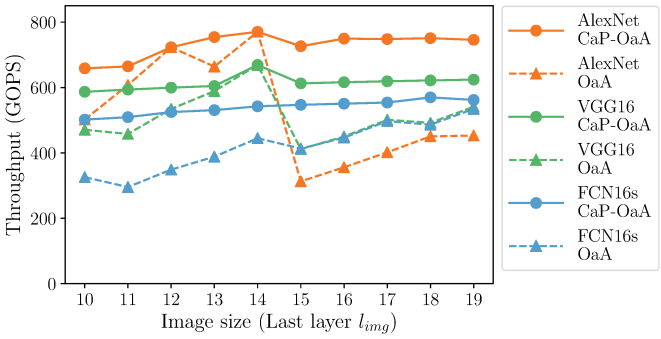


Figure 10: Throughput of AlexNet, VGG16 and FCN-16s

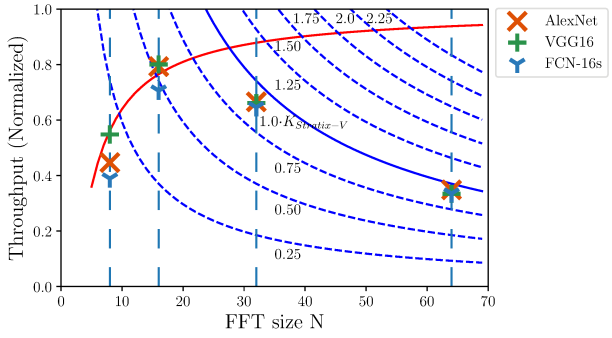


Figure 11: Actual throughput (normalized) for various N

[8] Jiantao Qiu, et al. 2016. Going Deeper with Embedded FPGA Platform for Convolutional Neural Network. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '16)*. ACM.

[9] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2012. Imagenet classification with deep convolutional neural networks. In *NIPS'12*.

[10] Andrew Lavin. 2015. Fast Algorithms for Convolutional Neural Networks. *CoRR* abs/1509.09308 (2015).

[11] Jonathan Long, Evan Shelhamer, and Trevor Darrell. 2014. Fully Convolutional Networks for Semantic Segmentation. *CoRR* abs/1411.4038 (2014).

[12] Yufei Ma, et al. 2017. Optimizing Loop Operation and Dataflow in FPGA Acceleration of Deep Convolutional Neural Networks. In *Proceedings of the 2017 ACM/SIGDA Intl. Symposium on Field-Programmable Gate Arrays (FPGA '17)*.

[13] Markus Püschel, et al. 2005. SPIRAL: Code Generation for DSP Transforms. *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"* 93 (2005).

[14] A. Podili, C. Zhang, and V. Prasanna. 2017. Fast and efficient implementation of Convolutional Neural Networks on FPGA. In *2017 IEEE 28th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*.

[15] Karen Simonyan and Andrew Zisserman. 2014. Very Deep Convolutional Networks for Large-Scale Image Recognition. *CoRR* abs/1409.1556 (2014).

[16] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott E. Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. 2014. Going Deeper with Convolutions. *CoRR* abs/1409.4842 (2014).

[17] Xianyi Zhang, et al. 2017. OpenBLAS. (2017). "www.openblas.net"

[18] Yufei Ma, et al. 2016. Scalable and modularized RTL compilation of Convolutional Neural Networks onto FPGA. In *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*.

[19] Hanqing Zeng, Ren Chen, and Viktor K. Prasanna. 2017. *Optimizing Frequency Domain Implementation of CNNs on FPGAs.* Technical Report. University of Southern California. http://ceng.usc.edu/techreports/2017/Prasanna%20CENG-2017-3.pdf

[20] Hanqing Zeng, Chi Zhang, and Viktor Prasanna. 2017. Fast Generation of High Throughput Customized Deep Learning Accelerators on FPGAs. In *2017 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*.

[21] C. Zhang and V. Prasanna. 2017. Frequency Domain Acceleration of Convolutional Neural Networks on CPU-FPGA Shared Memory System. In *Proceedings of the 2017 ACM/SIGDA Intl. Symp. on Field-Programmable Gate Arrays (FPGA '17)*.

[22] Jialiang Zhang and Jing Li. 2017. Improving the Performance of OpenCL-based FPGA Accelerator for Convolutional Neural Network. In *Proceedings of the 2017 ACM/SIGDA Intl. Symposium on Field-Programmable Gate Arrays (FPGA '17)*.