Optimizing CNN-based Segmentation with Deeply Customized Convolutional and Deconvolutional Architectures on FPGA

SHUANGLONG LIU, HONGXIANG FAN, XINYU NIU, HO-CHEUNG NG, YANG CHU, and WAYNE LUK, Imperial College London, UK

Convolutional Neural Networks (CNNs) based algorithms have been successful in solving image recognition problems, showing very large accuracy improvement. In recent years, deconvolution layers are widely used as key components in the state-of-the-art CNNs for end-to-end training and models to support tasks such as image segmentation and super resolution. However, the deconvolution algorithms are computationally intensive which limits their applicability to real time applications. Particularly, there has been little research on the efficient implementations of deconvolution algorithms on FPGA platforms which have been widely used to accelerate CNN algorithms by practitioners and researchers due to their high performance and power efficiency. In this work, we propose and develop deconvolution architecture for efficient FPGA implementation. FPGA-based accelerators are proposed for both deconvolution and CNN algorithms. Besides, memory sharing between the computation modules is proposed for the FPGA-based CNN accelerator as well as for other optimization techniques. A non-linear optimization model based on the performance model is introduced to efficiently explore the design space in order to achieve optimal processing speed of the system and improve power efficiency. Furthermore, a hardware mapping framework is developed to automatically generate the low-latency hardware design for any given CNN model on the target device. Finally, we implement our designs on Xilinx Zynq ZC706 board and the deconvolution accelerator achieves a performance of 90.1 GOPS under 200MHz working frequency and a performance density of 0.10 GOPS/DSP using 32-bit quantization, which significantly outperforms previous designs on FPGAs. A real-time application of scene segmentation on Cityscapes Dataset is used to evaluate our CNN accelerator on Zynq ZC706 board, and the system achieves a performance of 107 GOPS and 0.12 GOPS/DSP using 16-bit quantization, and supports up to 17 frames per second for 512x512 image inputs with a power consumption of only 9.6W.

CCS Concepts: • **Computer systems organization → Neural networks**; • **Hardware → Hardware accelerators**;

Additional Key Words and Phrases: FPGA, Convolutional Neural Networks (CNNs), Deconvolution, Hardware Acceleration, Segmentation

ACM Reference Format:
Shuanglong Liu, Hongxiang Fan, Xinyu Niu, Ho-Cheung Ng, Yang Chu, and Wayne Luk. 2018. Optimizing CNN-based Segmentation with Deeply Customized Convolutional and Deconvolutional Architectures on FPGA. *ACM Trans. Reconfig. Technol. Syst.* 1, 1, Article 1 (January 2018), 23 pages. https://doi.org/0000001.0000001

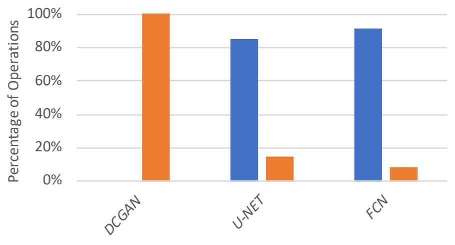
Authors' address: Shuanglong Liu, s.liu13@imperial.ac.uk; Hongxiang Fan, h.fan17@imperial.ac.uk; Xinyu Niu, xinyu.niu@corerain.com; Ho-Cheung Ng, h.ng16@imperial.ac.uk; Yang Chu, y.chu16@imperial.ac.uk; Wayne Luk, w.luk@imperial.ac.uk, Imperial College London, Department of Computing, London, SW7 2AZ, UK.

1936-7406/2018/1-ART1 $$15.00
https://doi.org/0000001.0000001

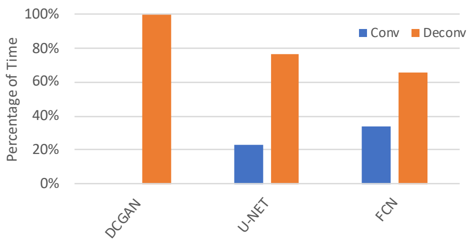
1 INTRODUCTION

Convolutional Neural Network (CNN) based algorithms have shown great performance improvement in lots of machine learning tasks including speech recognition [5], object detection [30] and image segmentation [20], achieving greatly improved accuracy or/and execution time [14, 20]. However, the CNN algorithms are very computationally intensive which becomes a major issue in their application to real time tasks on embedded devices. To address this problem, FPGAs have been widely adopted to accelerate these CNN algorithms [5, 27, 30], due to their highly-parallel bit-oriented architecture. In particular, FPGA-based accelerators achieve higher performance in terms of execution time compared to CPUs and consume much less power than GPUs while being more flexible and configurable than ASICs [27, 29]. Therefore FPGA platforms can provide real-time solutions for various image recognition tasks.

Deconvolution, also known as up-sampling or transposed convolution[1], has been widely used in the state-of-the-art convolutional neural networks (CNNs) and deconvolutional neural networks (DCNNs) [25] for computer vision applications such as scene segmentation [1], image denoising [28] and super-resolution imaging. Deconvolution performs a fundamentally new type of mathematical operator which aims to extrapolate new information from the input feature map. This contrasts with convolution that aims to interpolate the most relevant information from the input. Deconvolutional layers are mainly used in deep learning networks for two reasons: 1) convolutional sparse coding [25, 26]; 2) upsampling for image generation [14, 19], to support applications such as super resolution [22] and image-to-image translation [10]. Some popular examples of the state-of-the-art deconvlolution based networks include the deep generative models [21], U-Net [20] and Fully Convolutional Network (FCN) [14], just to name a few. There is a growing demand for FPGA-based deconvolution accelerators for real-time applications for two reasons: 1) Deconvolution is the only type of layer in DCNNs such as DCGAN [19] which doesn't have convolution and other layers, thus the main computational complexity comes from deconvolution operations in these networks (Figure 1a); 2) For CNNs consisting of both layers, deconvolution becomes the bottleneck in speed when convolution has been accelerated on FPGA[2] (Figure 1b).



(a)



(b)

Fig. 1. (a) The percentage of operations of convolution (*Conv*) and deconvolution (*Deconv*) layers. (b) The percentage of execution time when evaluated with the state-of-the-art performance normalized on Xilinx ZC706 board, where the estimated performance of *Conv* and *Deconv* are 187 GOPS (giga operations per second) and 10 GOPS respectively as presented in [18] and [29].

[1]While transposed convolution is commonly called "deconvolution" in the field of deep learning, it is actually not the inverse of convolution. For simplicity, the term "deconvolution" is used as transposed convolution in the rest of the paper.
[2]A detailed analysis is provided in Section 8.4.

Nevertheless, current FPGA-based CNN accelerators mainly focused on enhancing the performance of the convolutional layers in CNNs, and little research has put forward deconvolution accelerators on FPGAs. Several issues need to be addressed to design an FPGA-based deconvolution accelerator. First, there are different implementations of deconvolution algorithms, and a direct translation of CPU-optimized deconvolution algorithms to an FPGA will generally lead to inefficient designs. Therefore, a comparison and adaptation of current CPU-based deconvolution algorithms to FPGA platforms are needed. Second, a parametrized deconvolution design on FPGA is necessary to support different layers and networks and it is important to generate the deconvolution design optimized for performance and power on FPGA automatically when deploying different CNN networks. We tackle both problems in this work.

This work proposes an FPGA-based deconvolution accelerator for the state-of-the-art neural networks. We first propose efficient and highly customized architectures for the parametrized deconvolution algorithm and CNN algorithms. Then various optimization techniques such as memory sharing are performed to fully utilize the FPGA resources and improve the performance of the overall CNN accelerator. A non-linear optimization model together with the corresponding non-linear programming solver is proposed to generate the optimal deconvolution and CNN designs on FPGA automatically for any given CNN network.

A summary of the main contributions of this work are as follows:

- Parametrized and fully customized deconvolution architecture; This is the first work proposing efficient architecture for parametrized deconvolution layer implementation on FPGA by register-transfer level (RTL) development, to support semantic segmentation;
- An optimized CNN algorithm for real-time segmentation based on U-Net and a tailored architecture of CNN accelerator; The proposed CNN algorithm takes into account the efficiency of hardware implementation and achieves low latency with desirable accuracy; The CNN accelerator is optimized by sharing memory between CONV and DECONV modules;
- A non-linear optimization model for design space exploration; We introduce the performance model for the proposed CNN accelerator to find out the limiting resources and optimal design parameters by non-linear optimizations, and thus generate the optimal design automatically when deploying different CNN networks and FPGA devices;
- Evaluations of the proposed CNN accelerator on cityscapes dataset for real time image segmentation; Our system can process 17 frames per second for 512x512 image inputs with a power consumption only at 9.6W on Xilinx Zynq ZC706 board.

The rest of this paper is organized as follows: Section 2 provides some background knowledge on CNN and deconvolution algorithms. Section 3 presents the architecture of the parametrized deconvolution accelerator and our hardware implementation. Section 4 describes the architecture for the CNN algorithm and the optimization techniques. Section 5 introduces the performance model of the CNN accelerator and proposes the non-linear optimization model to explore the design space. Section 6 describes the workflow of the framework for automatic hardware mapping and code generation. Section 7 describes our proposed CNN algorithm based on the previous U-Net for real time image segmentation. Section 8 shows our experiment result of the proposed deconvolution accelerator and CNN accelerator. Section 9 gives a comparison between our implementation and existing work and Section 10 concludes this work.

2 BACKGROUND

2.1 CNN-based Image Segmentation

In the last decade, deep convolutional networks have outperformed the state-of-the-art in many visual recognition tasks. The typical use of convolutional networks such as LeNet, AlexNet and

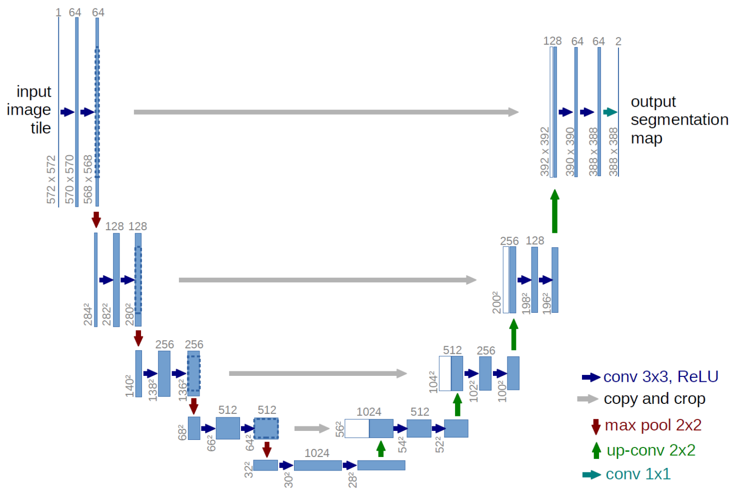


Fig. 2. The U-net architecture for Image Segmentation [20].

GoogleNet is on classification tasks such as object detection [20], where the output of an image is a single class label. However, in many recognition tasks such as segmentation, super resolution and image-to-image translation, the desired output should also include localization information, i.e., a class label is supposed to be assigned to each pixel in the image. The segmentation tasks are always more computationally intensive than the classification tasks. In CNN-based segmentation algorithms, the segmentation results are generally achieved by "upsampling" the input image using deconvolution layers in CNN.

A CNN typically consists of a sequence of computation layers stacked together [30]. These layers read input feature maps and generate output feature maps. For image segmentation, the input of the first layer is an input image and the output of the last layer is an image with the class labels assigned to each pixel. The layers used in CNN-based image segmentation algorithms are summarized as follows:

1. **Convolution layer** (*Conv*) is the core building block of a CNN that does the majority of the computation (Figure 1a). Convolutional layer convolves the input image or feature maps with the convolution kernel, producing one output feature map. There are often groups of different convolution kernel used in one layer and thus multiple filters of output feature map are produced.

2. **Pooling layer** is often inserted between successive *Conv* layers as a non-linear down-sampling to reduce feature map size and the computation for later layers. It applies a sliding window to the input and takes the maximum or average value as output of this region. They are called max pooling and average pooling respectively.

3. **Deconvolution layer** (*Deonv*) transforms the input in the opposite direction of a *Conv* layer but extrapolates new information from the input feature map. It is used as a mean to up-sample the input feature map towards the original input image resolution [16, 25].

One of the state-of-the-art CNN networks for image segmentation is U-Net proposed in [20] which won the ISBI Cell Tracking Challenge in 2015. The U-Net architecture is shown in Figure 2. It consists of 9 repeated two 3x3 *Conv* layers, each followed by a rectified linear unit (ReLU) and a 2x2 max pooling layer with stride 2 for downsampling. It also consists of upsampling of the feature map using 4 2x2 deconvolution layers. In the final layer, a 1x1 convolution is used to map the feature vector to the desired number of classes. In total the network has 23 layers. The deconvolution layers are crucial in the U-Net architecture for generating the whole image for segmentation.

2.2 Deconvolution Algorithms

Table 1. A summary of the parameters in the deconvolution layers in CNN.

Parameter	Layer	Description
H		Height of the input feature map
W		Width of the input feature map
H_O		Height of the output feature map
W_O		Width of the output feature map
N_C	*Deconv*	Number of channels in the input feature map
N_F		Number of filters in the output feature map
k		Height and width of the kernel
s		the stride
p		the amount of zero padding

Same as *Conv* layers, *Deconv* layers accept an input image of size $N_C * H * W$ and a group of coefficient matrix of shape $N_F * N_C * k * k$, then produce a volume of size $N_F * H_O * W_O$ where:

$$H_O = s * (H - 1) + k - 2 * p \tag{1}$$

$$W_O = s * (W - 1) + k - 2 * p \tag{2}$$

Table 1 summarises the parameters used in the deconvolution layer. Algorithm 1 describes the deconvolution layers in CNN in a high level which consists of the filter loop and channel loop. The implementation of the *deconv* operations on the two 2-D matrix shown in line 3 will be covered in detail in this section.

ALGORITHM 1: Deconvolution layer computation with two nested loops.

Input: input feature map I of shape $N_C * H * W$;
Input: A coefficient matrix K of shape $N_F * N_C * k * k$;
Output: output feature map of shape $N_F * H_O * W_O$;

```
1: for $f = 1$ to $N_F$ do
2:     for $c = 1$ to $N_C$ do
3:         $O[f] += deconv(I[c], K[f, c])$
4:     end for
5: end for
```

The state-of-the-art software (CPU and GPU) architecture of the *deconv* operation is implemented as transposed convolution, by adding appropriate amount of zero padding around and/or between the input feature map [4]. A deconvolution described by kernel size k, stride s, pad p and input size i is performed as a convolution with $k' = k, s' = 1, p' = k - p - 1$ and adding $s - 1$ zero padding between each input unit. This method is illustrated in Figure 3 where the white areas represent the added zeros to the original input map. Apparently, direct mapping of the CPU-optimized deconvolution algorithm onto FPGA will incur inefficient performance because this method involves adding many columns and rows of zeros to the input, resulting in a much less efficient computation on FPGA.

Therefore, we propose the use of an FPGA-efficient method by implementing *deconv* with the following four steps: (1) multiply a single input pixel by the $k * k$ kernel; (2) sum the results of step (1) where the outputs overlap; (3) repeat (1) and (2) for all input pixels; (4) remove the elements from output feature maps in the border of size p. This method is illustrated in Figure 4. Compared to

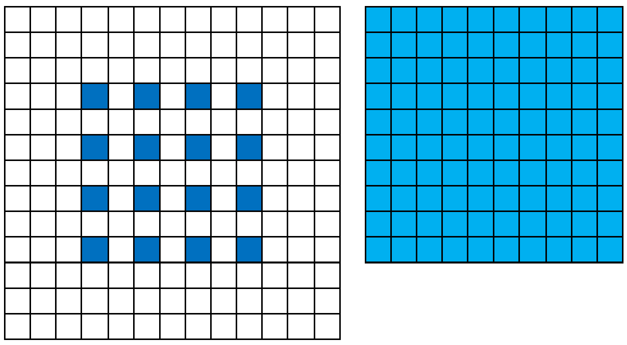


Fig. 3. Input map (*left*) to output (*right*) after *deconv* with $k = 4, s = 2, p = 0$ by doing a full 2-D convolution. Raw input size is 4x4 and output size is 10x10. Zero padded input size (which serves as input of *conv*) is 13x13.

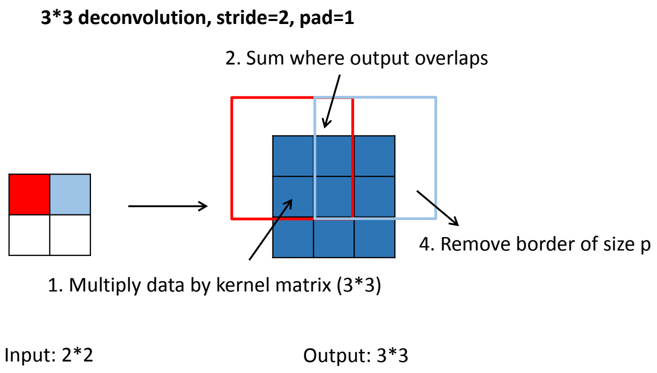

3*3 deconvolution, stride=2, pad=1
2. Sum where output overlaps
4. Remove border of size p
1. Multiply data by kernel matrix (3*3)
Input: 2*2
Output: 3*3

Fig. 4. Input map (*left*) to output (*right*) after *deconv* with $k = 3, s = 2, p = 1$ by 4 steps. Raw input size is 2x2 and output size is 3x3.

the transposed convolution implementation, this method has no requirement to add zeros between input map, thus it is more efficient for computation especially considering FPGA implementation. It should be noted that when $p = 0$, step 4 is no longer required and this case is considered in the next of the paper. When $s = k$, there is no overlap in the output of the multiplication result for each pixel, therefore step 2 is not needed and the computation can be as simple as to multiply each input pixel by the $k * k$ kernel. When $s < k$, the implementation of this method needs to deal with both column and row overlaps in the output. This will be described in detail in Section 3.

2.3 Algorithm Comparison

Here, we first provide a detailed analysis and comparison on the computational complexity of the two types of deconvolution algorithms, followed by a summary and comparison on the efficiency.

The computational complexity of CNN or one layer is represented by the total operations (OP) including multiplications and additions. For the CPU-optimized deconvolution, each channel computation needs $k^2 H_O W_O$ multiplications and $k^2 H_O W_O$ additions respectively. The total operations for one channel computation are:

$$OP_{CPU} = 2 * k^2 H_O W_O \tag{3}$$

The number of multiplications needed by the proposed algorithm for one channel is k^2HW (H and W are much smaller than H_O and W_O for deconvolution layers) and the number of additions needed by the overlapped area is $k(k-s)(H-1) + k(k-s)(W-1) + (k^2-s^2)(W-2)(H-2)$.

The total operations of the proposed algorithm are:

$$OP_{proposed} = k^2HW + k(k-s)(H-1) + k(k-s)(W-1) + (k^2-s^2)(W-2)(H-2) \qquad (4)$$

The deconvolution layers in the U-Net and FCN architectures are described in Table 2. The table also summarizes the number of operations (OPs) for the deconvolution layers of these two CNN architectures and the efficiency achieved by the proposed algorithm compared to the CPU version. For U-Net, the proposed algorithm achieves 4 times computational efficiency for each layer, and it achieves 83 times at best for FCN. We observe that the benefit is more significant when large kernel size is used for deconvolution layer. The reduced computational complexity can contribute to less execution time and higher energy efficiency given the same computational resources in FPGA.

Table 2. Number of operations for the deconvolution layers of U-Net and FCN.

CNN	layers	parameters	CPU-optimized OPs			Proposed OPs			Efficiency Speedup
		$(k, s, H\&W, N_C, N_F, H_O\&W_O)$	Multi	add	total	Multi	add	total	
U-NET	deconv1	(2,2,28,1024,512,56)	6.58G	6.58G	13.15G	1.644G	1.642G	3.28G	4x
	deconv2	(2,2,52,512,256,104)	5.671G	5.671G	11.34G	1.417G	1.415G	2.832G	4x
	deconv3	(2,2,100,256,128,200)	5.24G	5.24G	10.5G	1.31G	1.30G	2.60G	4.01x
	deconv4	(2,2,196,128,64,392)	5.04G	5.04G	10.07G	1.26G	1.24G	2.51G	4.02x
FCN	deconv1	(4,2,1,21,21,4)	0.113M	0.113M	0.226M	7.05K	12K	19.1K	11.8x
	deconv2	(4,2,4,21,21,10)	0.71M	0.71M	1.41M	0.113M	84.3K	0.197M	7.2x
	deconv3	(16,8,10,21,21,88)	0.874G	0.874G	1.75G	11.3M	9.69M	20.98M	83.4x

Besides the increased computational complexity, there are some other disadvantages of the CPU implementation. The following highlights the sources of these inefficiencies and summarizes the benefits of our approach:

- The zero-padding approach is inherently inefficient due to the additional unproductive arithmetic operations when using an FPGA, while the proposed approach avoids this inefficiency.
- Performing multiply-accumulate on the inserted zeros causes under-utilization of the computing resources available on FPGA.
- The inserted zeroes create different patterns of computation when sliding the convolution window (different kernel parameters between conv and deconv layers). As such, the same sequence of operations cannot be repeated across all the processing layers by using a single computational module, i.e., we need to instantiate multiple modules for different deconv layers besides the conv module. Nevertheless, our approach can easily be configured for extension and reuse with different parameters such as $(k, s) = (4, 2)$ and $(16, 8)$ in FCN, as will be shown in Section 3.

3 DECONVOLUTION HARDWARE DESIGN

The top level architecture of the deconvolution accelerator on FPGA is shown in Figure 5. Due to the limited on-chip memory resource on FPGA, all the processing data and coefficients of the net are stored in off-chip memories such as DDR. To process each deconvolution layer, the input map is first read from DDR to the *input buffer* which is implemented as RAM, in order to reuse the input data for filter processing as shown in line 1 in Algorithm 1. Correspondingly, the coefficient is cached in *coef buffer* which uses FIFOs with size of $k * k$, since the coefficients will only be used for computation once and won't be reused for filter processing as shown in Algorithm 1. Then one column of the coefficient kernel is read from the FIFOs and stored in registers, and sent to *deconv kernel* together with one single input data for multiplication. *deconv kernel* performs the *deconv* operation shown in line 3 in Algorithm 1 and is implemented based on the 4-step algorithm we mentioned above. Finally, before the output from *deconv kernel* is transferred to *output buffers*, accumulation is performed to sum the results of different channels, which is shown in the for loop in line 2 in Algorithm 1. It should be noted that ping-pong FIFO is used for *output buffer*, in order to process the next filter and transfer the current filter result back to DDR concurrently.

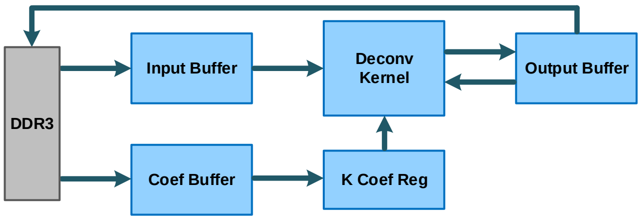


Fig. 5. The top level architecture of the deconvolution accelerator which implements the algorithm shown in Algorithm 1.

A parametrized deconvolution kernel design which implements the 4-step method in Section 2.2 is shown in Figure 6. This architecture supports different parameters of deconvolution layer, i.e. k, s, p, and therefore can be reused for different deconvolution layers and networks. For $s < k$, there are $k - s$ columns or rows overlapped between the output for two adjacent input data. Therefore a register array with size $k * (k - s)$ is used to deal with the overlap. Every cycle, one column of the coefficients kernel is read from FIFOs and multiplied by one single input data using k multiplies. The results are buffered in the $k * (k - s)$ register array and shifted every cycle. To deal with the column overlap, the results of the multiplies are added by the data in the last column of the registers before they are stored in the partial result buffer or output buffer. To deal with the row overlap, the outputs of the first $k - s$ adders are also added by the partial results before they are transferred to the output buffer. When $k = s$ there are no overlap computations.

The above architecture is customized to support different deconv layers that have different (k, s) configurations, and the kernels are optimized to improve hardware re-use thus achieving better efficiency and performance. This is explored by allowing in-kernel parallelism to the design. For example, the kernel of U-Net with $(k, s) = (2, 2)$ is used as two parallel computation data path for the 1*1 *conv* layer (the final layer) in U-Net, avoiding the introduction of another computation module to the FPGA. The FCN kernel is configured as $(k, s) = (16, 8)$ and its datapaths are used as 4 parallel kernels when computing *deconv* layers with $(k, s) = (4, 2)$. The performance is improved by 4 times in such situation with no extra cost.

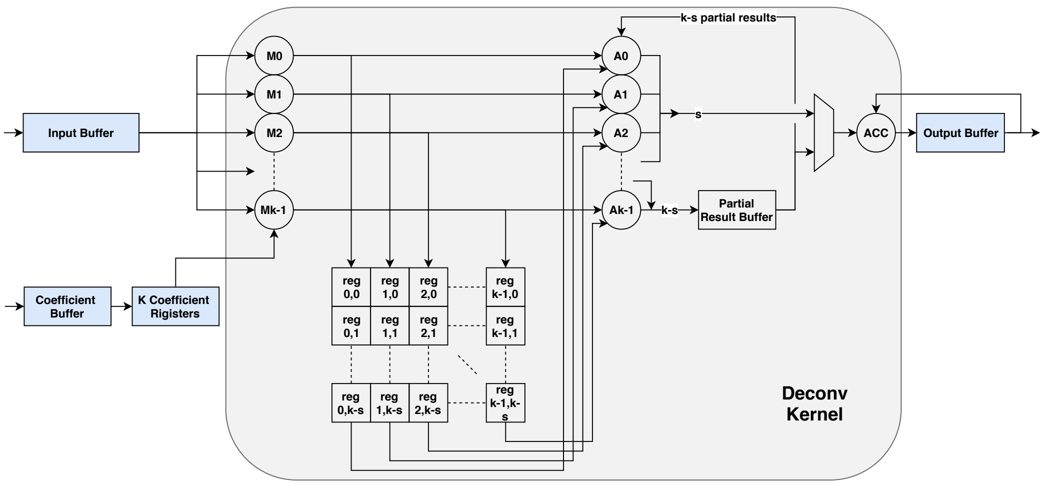


Fig. 6. Parametrized deconvolution accelerator design which supports different parameters of deconvolution layer and can be reused for different networks.

4 CNN ACCELERATOR DESIGN AND OPTIMIZATIONS

4.1 CNN Architecture

The architecture of the proposed accelerator on FPGA for CNN-based real-time image segmentation is shown in Figure 7. The proposed CNN accelerator design on FPGA is composed of several major components, which are the computation units (*Conv* and *Deconv* modules), on-chip memories, external memory and on-/off-chip interconnect. *Conv* and *Deconv* are the basic computation units for the CNN-based segmentation algorithms. All data for processing are stored in external memory. Due to the limitation of the on-chip memory size, data are first cached in on-chip buffers before being fed to the computation units. The AXILite Bus is used to connect the computation units on FPGA Program logic (PL) to the processor (PS) to configure the parameters of the layers.

The optimization to the CNN accelerator design focuses on fully utilising the existing hardware resource by the computational modules. The Deconv module is configured to perform different deconv layers to improve the efficiency as we mentioned earlier. The conv module and deconv module share the same input buffer as will be discussed later. However, it is necessary to introduce both conv and deconv modules for the segmentation accelerator. The fundamental difference in mathematical operations of these two modules leads to different patterns of data access and computation due to the different kernel sizes, sliding window and padding parameters. As such, the sequence of data access and operations cannot be repeated across all the processing layers by a single computation module which needs to perform both conv and deconv operations. Therefore, our proposed architecture has two modules in place (which is similar to the work in [18] having both conv and fc modules), and the computation kernels are optimized to improve hardware re-use and allow higher FPGA clock frequency thus better efficiency and performance.

4.2 Convolution Architecture

This section will briefly cover the convolution architecture since it's not the focus of this paper. The convolution architecture is based on the design in [30]. Since the pooling layer is always connected after *conv* layer, we implemented pooling layer inside *conv* layer by setting a pooling enable signal. To extend the *conv* layer to support the U-Net architecture, a crop operation is also added to the *conv*

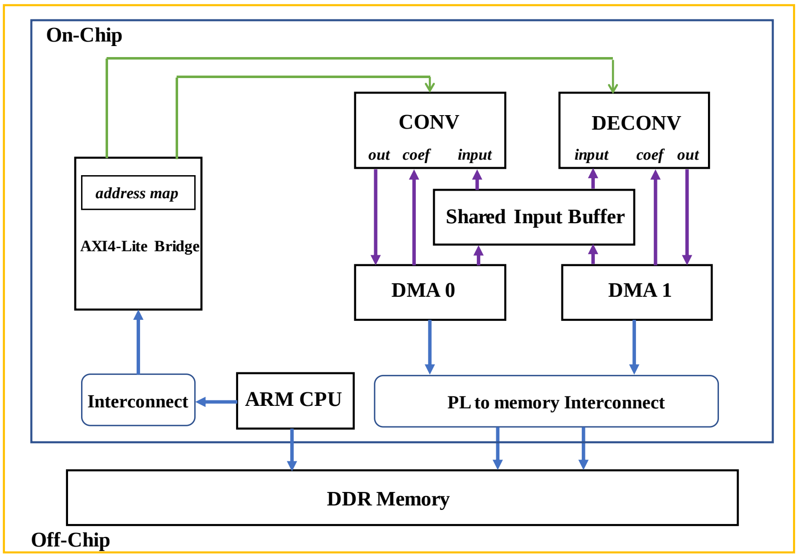


Fig. 7. The architecture of the FPGA-based CNN accelerator which integrates 2 computation units: CONV and DECONV.

layer by setting a crop enable signal. That is to say, by properly setting the pooling enable and crop enable signals, our convolution accelerator can actually support three combined functions: 1) single convolution; 2) convolution plus pooling; 3) convolution plus crop and convolution plus pooling. Double output buffer is also used in the convolution architecture to save the output transfer time.

4.3 Optimizations

In this section, we describe the optimization techniques used for the FPGA-based CNN accelerator in order to increase the system throughput. Computation and communication time are two main optimization directions [27].

Shared Memory. Due to the limited on-chip memory resources, the implementation of many machine learning algorithms on FPGAs have been shifted from computation bound to memory bound [12, 13]. The FPGA-based CNN accelerator is no exception and even more intense because of the large amount of processing data for each layer. In previous work in [29, 30] and other recent literature, the input buffers for different computation layers are separate, which increases the on-chip memory overhead and makes the memory resources particularly precious. However, the computation layers in current CNNs are following each other, which means they will never run concurrently. For this reason, we can share the input buffer as shown in Figure 7 between the *deconv* and *conv* modules. By this optimization, the input buffer size can be halved for the overall accelerator, which largely relieves the memory bottleneck of FPGA-based CNN accelerator.

Blocking Strategy. Because of limited memory resources on FPGA, input buffer cannot be large enough to hold the whole input map of one layer. Blocking is essential when implementing the CNN algorithms on FPGA. Blocking is to divide the input feature map into several parts to reduce input buffer usage, and it is applied to the original input feature map for each layer. Block input size mainly depends on the available memory resource and the shared input buffer size. In this work, we optimize the block input size as it affects the overall communication time.

Parallel Computation. Other standard optimization techniques include parallel computation and data quantization. In this work, we fully parallelize the kernel matrix computation in *conv* and *deconv* layers since the kernel size is generally small (e.g. $k = 3, 5$ for *conv* and $k = 2, 4$ for *deconv*). We explore the design space in the aspects of data parallelism (Pv) and filter parallelism (Pf) [3]:

- Data parallelism (Pv). Pv represents the amount of input data processed in parallel. While computing matrix multiplication, it is possible to compute multiple kernels in parallel for a number of input data. This level of parallelisation can be achieved by reading multiple input in each cycle and replicating computation kernels Pv times. However, Pv is limited by the block size and memory bandwidth.
- Filter parallelism (Pf). Pf represents the number of filters processed in parallel, which has an upper bound N_F. It should be noted that unlike the data parallelism (Pv), filter parallelism requires replicating both computation kernel and output buffers and thus it increases both memory and logic overheads. Nevertheless, due to the blocking strategy, Pv is limited by the block height or width, so Pf is necessary to further improve the performance.

Reduced Precision. Many recent research has shown that CNNs are robust to low bitwidth quantization. Instead of using the default double or single floating point precision in CPU, fixed-point precision can be used in FPGA-based CNN accelerator to achieve an efficient design optimized for performance and power. In this work, we use multiple fixed-point data format such as 32 bits, 24 bits and 16 bits to implement our proposed design, in order to allow optimizations for high parallel degrees mentioned in the above section. We evaluate the performance and power efficiency of the proposed CNN accelerator for real time image segmentation in Section 8.

4.4 Design Parameters Tuning

The idea is to tune the design parameters mentioned above and have a trade-off between the memory and computation resources on FPGA to achieve the optimal performance for the proposed CNN accelerator. Given a specific data width, we first need to determine the input block size based on the available memory resource for a given FPGA device; Then we can allocate the memory to input buffers for sharing between the computation layers, and output buffers for each computation unit; Based on the memory usage and computation resources (e.g. DSPs), we need to adjust the value of filter parallelism, i.e. Pf and data parallelism Pv for each computation module. This process involves co-optimizing for the resource allocation between *conv* and *deconv* modules, in order to achieve highest DSP utilization over time thus the highest processing speed of the system. The details on how to solve out the optimal values of the input block size, Pv and Pf are discussed in the next Section.

5 PERFORMANCE MODEL AND DESIGN SPACE EXPLORATION

5.1 Performance Model

This section provides a detailed analysis of the performance of our proposed accelerator using the above optimizations. In both *conv* and *deconv* modules, we design the kernel such that for a single pipeline, one result is generated each cycle. For each computation layer, the execution time includes: (1) loading data from off-chip memory into input buffer; (2) finishing computation for all the computations of involved channels and filters; (3) writing computation results back into off-chip memory. Here we first define a few terms used in our performance model in Table 3.

Correspondingly, the execution time in terms of clock cycles for computing a single filter includes:

[3]Channel parallelism is not considered in this work as it affects the overall architecture.

Table 3. Terms definition in the performance model.

Parameter	Definition
Pv	Data-level parallelism
Pf	Filter-level parallelism
Freq	Clock frequency
BWi	Memory bandwidth for loading data into kernel
BWo	Memory bandwidth for writing data into kernel
DW	Data width
B_H	Height of the blocked input
B_W	Width of the blocked input

1. Load time i.e., time to load N_C channels into cache:

$$T_1 = \frac{N_C * W * H * DW}{\text{Freq} * BWi} \tag{5}$$

2. Computation time i.e., time to compute all channels for this filter:

$$T_2 = \frac{N_C * W * H}{Pv} \tag{6}$$

3. Write time i.e., time to write the filter result into memory:

$$T_3 = \frac{W_O * H_O * DW}{\text{Freq} * BWo} \tag{7}$$

Therefore the total execution time to compute the result for a filter inside the layer is:

$$T_{\text{filter}} = \frac{N_C * W * H * DW}{\text{Freq} * BWi} + \frac{N_C * W * H}{Pv} + \frac{W_O * H_O * DW}{\text{Freq} * BWo} \tag{8}$$

When we are computing all the filters in the layer (normally, convolution and deconvolution layers have more than 1 filter), there would be two major savings:

- There is no need to load the data to cache again as all the filters compute on the same data;
- The data write time can be overlapped with computation time via the double output buffer;

Therefore, the overall execution time of one computation layer is:

$$T_{\text{layer}} = \frac{N_C * W * H * DW}{\text{Freq} * BWi} + \frac{N_C * N_F * W * H}{Pv * Pf} + \frac{W_O * H_O * DW}{\text{Freq} * BWo} \tag{9}$$

Only the computation time is increased by $\frac{N_F}{Pf}$ times, and the other two parts stay the same because the data are reused and output double buffer hides the write latency (one filter of data transfer after all the computation is still necessary). The total execution time of the whole network is the sum of the time of each layer.

5.2 Design Space Exploration

Given a specific CNN algorithm i.e. number of layers and N_C, N_F for each layer, the design space exploration is to find the optimal design parameter configuration i.e. block size ($B_H * B_W$), Pv and Pf, under the available logic and memory resources in the target FPGA device, in order to minimize the total execution time for real-time applications.

There are two major limiting resources for FPGA-based CNN accelerator: one of them is the logic resources such as LUTs and DSPs; the other is the memory resource i.e. the number of BRAMs. We model the resources usage of our design for a given data-width as follows:

Logic resources. It covers the usage of LUT, FF and DSP, which is simply linear with respect to $Pv * Pf$. Based on [30], DSPs are the limiting resource for the computation kernels, so only DSP usage is considered in this category. Since we fully parallelize the matrix computation in *conv* and *deconv* kernel, LUTs are used to implement fixed-point adders in order to save DSPs, as the adders consume much fewer LUTs compared to that of multipliers and considering the available LUTs are far more than the DSPs on FPGA. Let DSP^M represents the number of DSP usage of one multiplier, the total DSPs of the CNN accelerator are:

$$\begin{aligned} DSP^{\text{Net}} = & \, DSP^M * k^{\text{conv}} * k^{\text{conv}} * Pv * Pf \\ & + DSP^M * k^{\text{deconv}} * k^{\text{deconv}} * Pv * Pf \end{aligned} \tag{10}$$

Memory resource. The memory resources are mainly occupied by the shared input buffer and output buffer for each computation kernel (*conv*, *deconv* and pooling). For input buffer, it needs to store the whole input block of total channels for data reuse, and the size is $N_C^{max} * B_H * B_W$, where N_C^{max} is the maximum value of all the channels in the CNN algorithm. The output buffer size are $B_H * B_W$, $k * k * B_H * B_W$ and $B_H * B_W/4$ for *conv*, *deconv* and pooling layers respectively. When considering double buffers for output and filter parallelism, each output buffer size needs to increase by $2 * Pf$ times. Kernel coefficient has no need to be cached in RAMs because they are not reused during computations, therefore only a small FIFO of size $k * k$ can be implemented to buffer the streaming of kernel coefficient from DMA. Therefore, the total BRAMs used for our design can be estimated as:

$$\begin{aligned} BRAM^{\text{Net}} = & \frac{Nc^{max} * B_H * B_W * DW}{BRAM_{\text{size}}} + \\ 2 * Pf * & \frac{(B_H * B_W + k * k * B_H * B_W + B_H * B_W/4) * DW}{BRAM_{\text{size}}} \end{aligned} \tag{11}$$

Based on the above performance model and resource usage estimation, we need to obtain the design parameters to minimize the latency of the proposed CNN accelerator for the whole net computation. The whole optimization process is to determine the largest block size we can buffer in the device and the parallelism i.e., resource allocation between *conv* and *deconv* modules. Therefore this process involves co-optimizing between conv and deconv layers and achieving best efficiency of resource utilization (DSPs). The goal is to reduce the whole execution time of the evaluated network. Please note that the optimization goal is the latency instead of the accuracy, so the trade-off between data-width and accuracy is not explored in this work although the accuracy of the network for different data-width is discussed in Section 7. That is to say, our design space is explored by taking the data-width as an input, and we use a set of fixed-point data widths such as 32 bits, 24 bits and 16 bits to evaluate our design.

In summary, the design space exploration is to solve the nonlinear optimization problem as follows:

$$\begin{aligned} \underset{B_H, B_W, Pv, Pf}{\text{minimize}} \quad & \text{sum of time } T \text{ } in \text{ (9) for each layer in net} \\ \text{subject to} \quad & DSP^{\text{Net}} \leq \text{total DSP}, \\ & BRAM^{\text{Net}} \leq \text{total BRAM on FPGA}. \end{aligned}$$

The objective function which is the total execution time contains the information of the network i.e. H, W, N_f of each layer. The resource modelling is under a given data-width. Based on this optimization model, the Nonlinear programming solver *fmincon* in Matlab Optimization Toolbox is utilized to solve this optimization problem. Therefore, our tool automatically generates the optimal design parameters before we configure the FPGA device for final run, and then implement the optimal design in the target device, for any given CNN algorithm (such as U-Net, FCN).

6 AUTOMATIC HARDWARE MAPPING FRAMEWORK

An automatic hardware mapping framework is developed to automatically generate the low-latency hardware design for a given CNN model on the target device. Figure 8 shows the work-flow of the framework. The inputs are the CNN model parameters for each computational layer (such as N_C, N_F, maximum input image size), and the resources of the target device (BRAMs,DSPs and DDR memory bandwidth). Some other constraints are defined by the user such as the required execution time or the accuracy. The framework works through three steps: 1) first, the Matlab tool *fmincon* mentioned above is used to find out the optimal design parameters such as block size, parallelism: Pf and Pv, and data-width to satisfy a given time constraint; 2) the software C++ code is generated based on the design parameters and is run to set the block size and configure the parameters of the layers; 3) synthesizable Verilog hardware code is generated based on the parallelism and memory configuration information from the first step.

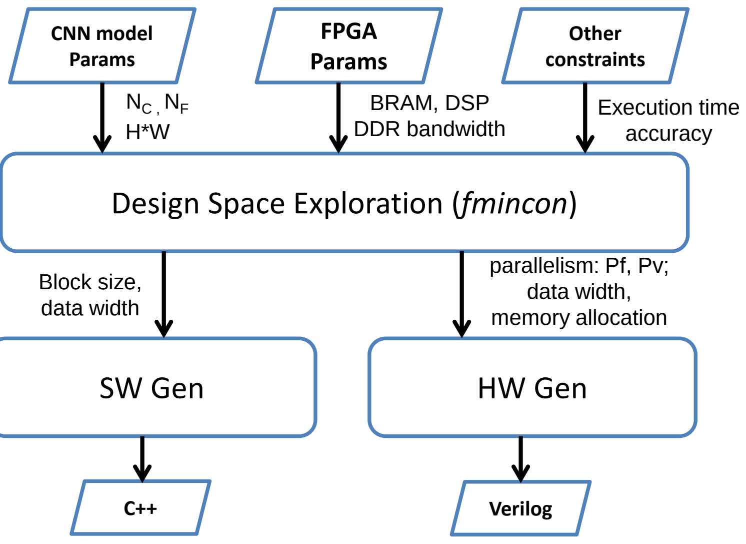


Fig. 8. The framework for automatic hardware optimization and mapping, and code generation for any given CNN and FPGA device.

The first step for design space exploration is using the Matlab tool, and code generation tool is implemented in Python. Based on the framework, we can easily extend our hardware to support different CNN architectures and applications such as classification, object detection, and segmentation. The framework largely improves design quality and designers' productivity by automatically generating the optimal hardware design for any CNNs before we configure the FPGA device for the final run.

7 PROPOSED CNN ALGORITHM FOR REAL-TIME SEGMENTATION

Image segmentation is the process of assigning a label to every pixel in an image such that pixels with the same label share certain characteristics such as color or intensity. The result of image segmentation is a set of segments that collectively covers the entire image. Each of the pixels in a region is similar with respect to some characteristic. One of the state-of-the-art networks for image segmentation is the U-Net architecture proposed by Olaf Ronneberger et. al. in 2015 [20]. Segmentation of a 512x512 image takes less than a second on a Nvidia Titan GPU. In this work, we evaluate this network architecture on the Cityscapes Dataset [3] which focuses on semantic understanding of urban street scenes.

For real-time application, it requires to assign a class label to each pixel in the input image at a very high speed and therefore the speed or the real-time response of the network is more important than the accuracy i.e., the segmentation result. For this reason, we propose an optimized U-Net CNN architecture based on the original one through three general model compression methods: 1) channel prunning [8]; 2) depth (number of layers) reduction [7] and 3) resolution multiplier [6, 9]. The configurations of each layer of the optimized U-Net architecture are described in Table 4. In total, it has 23 layers: eighteen 3*3 *conv* layers, four 2*2 *deconv* layers and one 1*1 *conv* layer. Three modifications and improvements are performed to improve the hardware efficiency of this architecture compared to the original one: 1) the number of filters of each layer is reduced, in order to save computation time; 2) the crop is removed and only copy needs to be performed; 3) unpadded convolutions are changed to padded for each *conv* layer, in order to keep the same output size as input for simple hardware implementation.

Table 4. Configurations of the proposed U-Net architeture.

Layer	1	2	3	4	5	6	7	8	9	10	11	
kernel	3*3 conv	3*3 conv 2*2 pool	3*3 conv	3*3 conv 2*2 pool	3*3 conv	3*3 conv 2*2 pool	3*3 conv	3*3 conv 2*2 pool	3*3 conv	3*3 conv	2*2 deconv	
Input	512*512	512*512	256*256	256*256	128*128	128*128	64*64	64*64	32*32	32*32	32*32	
N_C	3	8	8	16	16	32	32	64	64	128	128	
N_F	8	8	16	16	32	32	64	64	128	128	64	
Layer	12	13	14	15	16	17	18	19	20	21	22	23
kernel	3*3 conv	3*3 conv	2*2 deconv	3*3 conv	3*3 conv	2*2 deconv	3*3 conv	3*3 conv	2*2 deconv	3*3 conv	3*3 conv	1*1 conv
Input	64*64	64*64	64*64	128*128	128*128	128*128	256*256	256*256	256*256	512*512	512*512	512*512
N_C	128	64	64	64	32	32	32	16	16	16	8	8
N_F	64	64	32	32	32	16	16	16	8	8	8	1

The optimized U-Net architecture reduces the computational complexity in terms of operations from 30 GOP (giga operations) of the original one to 5.92 GOP while achieving the pixel-level segmentation accuracy of 60.8% compared to around 68% of the original network on Cityscapes Dataset. Since our proposed algorithm supports pixel-level segmentation, its functionality is not evaluated in terms of classification accuracy. The accuracy is measured based on the metric of class-level Semantic Labeling scores on cityscapes dataset. 60% of semantic labeling score indicates that 60% of all pixels in the analysed image are correctly labelled. For a 1080p-resolution image, this means 1.2 million pixels are correctly labelled. Under such accuracy, most of the semantics in the image can be understood. The official criterion on this dataset includes two levels, i.e., class

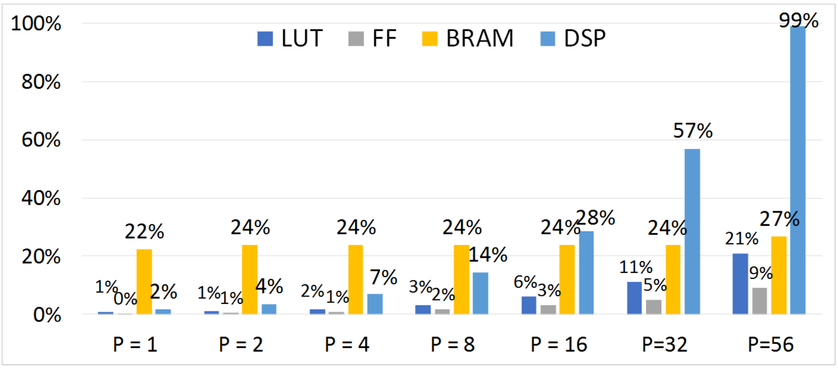


Fig. 9. Resource utilisation vs. Parallelism ($P = Pv * Pf$) of the deconvolution accelerator in ZC706 board.

and category [2]. The class-level score is much smaller than category-level score, since the number of classes is more than the number of categories in this dataset. One of the state-of-the-art CNN algorithms FCN-8s has a class-level score of 57.4%, which is smaller than our score. Among all the algorithms recorded in the official website, our design at a class-level score of more than 60% is beyond or comparable to most previous networks [2].

8 EVALUATION

8.1 Implementation Details

The proposed deconvolution accelerator, together with convolution accelerator are developed using Verilog HDL. Fixed point arithmetic operators are generated using Xilinx IPs with arbitrary bitwidth precision. The hardware system shown in Figure 7 is built on Xilinx Zynq ZC706 board which consists of a Xilinx XC7Z045 FPGA, dual ARM Cortex-A9 Processor and 1 GB DDR3 memory. The choice of FPGA is based on the consideration of area and power consumption. The whole system is implemented with Vivado Design Suite. The FPGA XC7Z045 is programmed with our deconvolution and CNN accelerator respectively. The ARM processor is used to initialize the accelerator, set the layer parameters and transfer the data for each layer. An overview of the implementation block diagram is shown in Figure 7. All designs run on a single 200 MHz clock frequency. The DDR3 memory in our design has a datapath width of 64 bits and it operates at the same clock frequency (200 MHz) as the FPGA engine. All results are post place and route except if it is stated otherwise.

For comparison, the respective software implementations run on CPU and GPU are using the deep learning software framework Caffe [11] in CentOS 7.2 operating system. The CPU platform is Intel Core i7-950 CPU@3.07GHz (8 cores). The GPU platform is Nvidia TITAN X (Pascal) (3840 CUDA cores with 12GB GDDR5 384-bit memory).

8.2 Deconvolution Accelerator Evaluation

We first evaluate the proposed deconvolution accelerator[4] for 32-bit data quantization. Figure 9 shows the resource utilization for the parametrized deconvolution accelerator with different parallelism ($P = Pv * Pf$). From this Figure, we can see when the parallelism is small, BRAMs dominate the resource usage; when parallelism increases, the DSPs usage increases a lot and becomes the limiting computation resource. This is expected as the main memory usage is the

[4]Deconvolution accelerator only integrates the *deconv* module in FPGA program logic while CNN accelerator implements both *conv* and *deonv* modules on FPGA. Therefore, these two accelerators can have different degree of parallelism and resource usage.

Table 5. Deconvolution Accelerator on FPGA vs. GPU and CPU implementations.

	FPGA	GPU	CPU
Platform	ZC706	Nvidia Titan X (Pascal)	Intel Core i7-950
Compiler	Vivado (2016.2)	Caffe (CUDA 8.0)	GCC (8 cores)
Compile Flags	-	-Ofast	-O3
Clock	200 MHz	1531 MHz	3.07 GHz
Technology	28 nm	16 nm	45 nm
Precision	32-bit fixed-point	32-bit floating-point	32-bit floating-point
Deconv time in U-Net	125ms	37.11ms	517ms
Power	9.60W	168W	106W
Energy	1.20J	6.24J	54.80J
Deconv time in FCN	0.25ms	0.1144ms	2.37ms
Power	9.60W	147W	106W
Energy	0.0024J	0.0168J	0.2512J

[1] CPU's power consumption is measured at the wall by a power meter when running the application.
[2] GPU's power consumption is recorded using NVIDIA System Management Interface (nvidia-smi).
[3] FPGA's power consumption is obtained from the board using a power meter.

shared input buffer which has no relationship with the parallelism. For the output buffer, only Pf has an effect on the BRAM usage. However, the DSP usage is linear to $Pv * Pf$, making it the main bottleneck in terms of computation resources when the degree of parallelism is high.

To evaluate the parametrized architecture, we test two cases of the deconvolution layers: 1) *deconv* in original U-Net which has parameters: $k = s = 2, p = 0$; 2) *deconv* in FCN which has parameters: $(k, s) = (16, 8), (4, 2)$ and $p = 0$. Therefore the deconvolution layers in FCN require the sum of overlapping part of the output results. The performance comparison between our accelerator and the respective software implementation on CPU and GPU is summarized in Table 5. The CPU-based system was developed utilizing all the available cores in the system (i.e. 8), as well as using -O3 optimizations. Compared to CPU design, our accelerator achieves 4.14x to 9.48x speedup, and consumes much less power. Nevertheless, GPU implementations beat the FPGA implementation on runtime. It is not surprised because of the current small number of DSPs (900) in the target device. However, our accelerator has much less power consumption and better energy efficiency than GPU.

Table 6. Proposed Deconvolution Accelerator vs. previous design on FPGAs.

	Chip	Precision	#DSP	Frequency	GOPS	GOPS/DSP
[29] in 2017	XC7Z020	12-bit fixed-point	220 (95% used)	100 MHz	2.6	0.012
Ours	XC7Z045	32-bit fixed-point	900 (99.5% used)	200 MHz	90.1	0.1

Finally we compare our deconvolution accelerator's performance with existing deconvolution accelerator in [29]. The results are shown in Table 6. Our deconvolution accelerator achieves 90.1 GOPS (giga operations per second) and 0.1 GOPS/DSP for 32-bit fixed point precision. Our work outperforms the design in [29] significantly which used 12-bit fixed point, and our deconvolution accelerator achieves around 34.7x improvement in terms of GOPS and 8.3x for GOPS/DSP.

8.3 CNN Accelerator Evaluation

In this section, we evaluate the performance of our CNN accelerator for the optimized U-Net architecture shown in Table 4, under multiple data quantizations. It should be noted that there is no obvious accuracy loss in the image segmentation result when reducing the precision from 32-bit floating point to fixed-point 16-bit quantization version for the inference process, as long as the training stage is using 32-bit floating point. That is to say, even using 16-bit fixed point precision, we still achieve the pixel-wise segmentation accuracy of 60.8% for image segmentation on Cityscapes Dataset.

We first use the non-linear optimization model to automatically generate the optimal CNN design for the optimized U-Net under a given data quantization on the target device. Table 7 summaries the optimal design parameters for a set of fixed-point data width and it also shows BRAM and DSP utilization. The execution time of the optimized U-Net was measured from running the optimal design on FPGA board for one 512*512 image segmentation process. The total number of BRAMs and DSPs in ZC706 are 545 and 900 respectively. For each candidate of data width, DSP is the limiting resources and it limits the parallelism we can achieve in the target device. The optimal performance we can obtain is 0.058s for 512*512 input image under 16-bit quantization, which is around 17 frames per second. Since double buffer is used to transfer data, the majority of run time is spent on computation in the case of our system.

Table 7. Optimal design parameters and its corresponding resource utilization and measured performance in Xilinx ZC706 FPGA board.

DW/bit	Block Size	Pf^{conv}	Pv^{conv}	Pf^{deconv}	Pv^{deconv}	DSP usage	BRAM usage	runtime (s) on board
32	32*32	2	8	1	8	92.44%	30%	0.141
24	64*64	4	8	1	16	78%	85%	0.087
16	64*64	4	16	1	16	71.1%	66.6%	0.058

As a design reference, we generate the optimal design parameters from our model when deploying to a bigger device Zynq XC7Z100 which has 755 BRAMs and 2020 DSPs in total. The results are shown in Table 8. As can be seen, when data width is 24-bit or 16-bit, the computational bottleneck is the memory resource instead of DSPs (which is the case when deploying to Xilinx ZC706). When DW = 16-bit, the memory limits the block size we can process and thus increases communication overhead; when DW = 24-bit, the memory limits filter parallelism as shown in the table that the optimal BRAM usage already exceeds DSP usage. When deploying to this device, with 16-bit quantization, the system is expected to process up to 25 frames for 512*512 image per second.

Table 8. Optimal design parameters and its corresponding resource utilization in Zynq XC7Z100 FPGA device.

DW/bit	Block Size	Pf^{conv}	Pv^{conv}	Pf^{deconv}	Pv^{deconv}	DSP usage	BRAM usage
32	64*64	2	16	1	8	69.7%	90.5%
24	64*64	4	16	1	16	63.4%	70%
16	64*64	4	32	1	16	60.2%	49%

The final system is implemented with 16-bit quantization in Xilinx ZC706 board and its detailed resource utilization is shown in Table 9. We compare the proposed CNN accelerator design with the respective optimal CPU (8 cores) and GPU implementations evaluated on the optimized U-Net. The results are summarized in Table 10. The platform and software implementation configurations are same as Table 5. Our CNN accelerator achieves a speedup of 10x in processing speed and 110x improvement in energy efficiency compared to the CPU design using all available threads. Compared to the state-of-the-art implementation on the world's most powerful GPU (Titan), our accelerator is a bit slower but achieves an improvement of 8x in energy efficiency.

Table 9. The resource utilization of the final design in ZC706 FPGA under parameter configuration: precision is 16-bit fixed point, block size is $64 * 64$, Pf and Pv of conv and deconv are (4,16) and (1,16) respectively.

Resource	LUT	FF	BRAM	DSP
Utilization	85679	110643	364	640
Total	218600	437200	545	900
Percentage(%)	39	25	67	71

Table 10. CNN accelerator on FPGA vs. GPU and CPU implementations evaluated on the optimized U-Net architecture.

	FPGA ZC706	GPU Titan X	CPU i7 (8 cores)
Optimized U-Net execution time (ms)	58.4	26.2	574
Power(W)	9.60	168	106
Energy (J)	0.56	4.40	60.9

Our system achieves the frame rate of 17 fps. The optimized U-Net network has 5.9 GOP (giga operations) in total including multiplications and additions. Our CNN accelerator achieves an average performance of 107 GOPS and resource efficiency of 0.12 GOPS/DSP in ZC706 device. We compare our CNN accelerator with the previous state-of-the-art in Table 11. Our accelerator achieves the lowest power consumption, and highest performance and resource efficiency of deconvolution accelerator in terms of GOPS/DSP. Compared to [18], our convolution computation engine and overall CNN accelerator provide comparable performance in terms of GOPS/DSP. It should be noted that: designs in [27] were designed for convolution layers only, those in [29] were designed only for deconvolution layer and [18] implemented convolution and fully-connected layers. In this work, we implemented both convolution and deconvolution layers in our proposed CNN accelerator which has never been done and evaluated on FPGA before, to the best of our knowledge.

8.4 Discussions

In terms of GOPS of the overall CNN accelerator consisting of both CONV and DECONV modules, it is $(r + 1)/(r + t)$ times in terms of GOPS of CONV, i.e.,

$$GOPS_{\text{Net}} = \frac{r+1}{r+t} GOPS_{\text{CONV}} \qquad (12)$$

Table 11. Our CNN accelerator vs. previous FPGA accelerators.

	[27] in FPGA15	[18] in FPGA16	[29] in 2017	Ours
Platform	Virtex7 VX485T	Zynq ZC7045	Zynq ZC7020	Zynq ZC7045
Clock (MHz)	100	150	100	200
Precision	32-bit float	16-bit fixed	12-bit fixed	16-bit fixed
#DSP	2800	900	220	900
Power (W)	18.61	9.63	-	9.60
Performance (GOPS)	61.62 (CONV)	187 (CONV) 137 (overall)	2.6 (DECONV)	125 (CONV) 29 (DECONV) 107 (overall)
Resource Efficiency (GOPS/DSP)*	0.022 (CONV)	0.207 (CONV) 0.15 (overall)	0.012 (DECONV)	0.14 (CONV) 0.033 (DECONV) 0.12 (overall)

* The total number of DSP in device is used to calculate GOPS/DSP for each module and overall when evaluating CNN accelerator, to be consistent with [18] and [29].

where r is the ratio of GOP of CONV and DECONV in a specific CNN algorithm and t is the ratio of GOPS of CONV and DECONV modules. For example, the optimized U-Net has 5.6 GOP for CONV and 0.3 GOP for DECONV, thus $r = 18.7$; our CNN accelerator achieves $t = 4.31$. Based on Equation (12), when DECONV consumes a large percent of operations in CNN (i.e., r is small[5]), or when DECONV has less performance (GOPS) than CONV on FPGA (i.e., t is large), the overall performance of the CNN accelerator is largely reduced. This is exactly the motivation of this work which focuses on the acceleration of deconvolution on FPGA. The relationship in (12) also provides a guide to tune the performance of CONV and DECONV in FPGA for a given CNN network. Previous work such as [18] didn't support deconvolution module and thus most computational resources on FPGA were spent on the CONV module. Therefore, it has higher GOPS/DSP performance of the CONV module. We integrate both CONV and DECONV modules in our CNN accelerator. As a consequence, the proposed design supports semantic segmentation, while the previous work only supports image classification. Nevertheless, our accelerator achieves a processing speed of 17 fps on the same device, although segmentation is a more complex task compared to classification task. The main benefits of our design come from that 1) algorithm optimization i.e., reducing the complexity of network; 2) high-performance deconv design with prior state-of-the-art conv design; 3) co-optimizing between the conv and deconv modules.

To the best of our knowledge, this is the first hardware accelerator architecture to support semantic segmentation deep-learning networks. Compared with CPU and GPU implementation, we achieve 110x and 8x improvement of energy efficiency respectively. The system can process up to 17 frames per second for 512*512 image segmentation and therefore it is very suitable for real time embedded applications.

[5] r is around 6 for the U-Net in [20] and $r = 0$ for DCGAN in [19] as all the layers are DECONV.

9 RELATED WORK

There is numerous work related to FPGA-based CNN accelerators, which mainly focus on the acceleration of convolution layer. [27] proposed the roofline model to analyse the accelerator's computing latency and required memory bandwidth. However this work only optimized convolution layer in CNN architecture. Besides, this work is based on Vivado HLS implementation and is hard to tune the design parameters. [18] optimized CNNs on embedded FPGA platform by dynamic-precision data quantization to reduce memory footprint and bandwidth requirements. [30] optimized the CNN algorithms for object detection on FPGA. [17] introduced the Sparse CNN (SCNN) accelerator architecture that exploited both weight and activation sparsity to improve both performance and power. Yet all these works are focused on the convolution and fully-connected layers, and they didn't evaluate the networks with deconvolution layers to support image segmentation. The state of the art convolution accelerator on FPGA is proposed in [15] by implementing convolution layers using a fast and efficient method: Winograd minimal filtering algorithm. [15] outperformed all previous work on convolution acceleration. However, Winograd algorithm is only efficient for very small kernel size like 3*3 and therefore it cannot be used in very general CNNs. For this reason, we didn't compare our work to [15] since our CNN accelerator is parametrized and can be used for most state of the art CNNs.

Recent FPGA-based accelerators for deconvolutional networks are presented in [23, 24, 29]. Yazdanbakhsh *et al.* [23, 24] proposed an end-to-end FPGA accelerator for GANs that combined MIMD and SIMD models while separating data retrieval and data processing units at the finest granularity. However, their designs in [23, 24] are based on the transposed convolution implementation and therefore are computationally inefficient compared to the method here as we mentioned earlier. Zhang *et al.* [29] proposed an accelerator with Vivado HLS tool and provided statistical analysis to find out the most cost-efficient bitwidth. However, this work only implemented deconvolution layer and can only be applied for deconvolutional neural networks (DCNNs). Our implementation takes advantage of deconvolution algorithm adaptation, shared memory for CNN accelerator and balance between logic and memory resource usage. Therefore our approach outperforms their work significantly.

The main difference of our work compared to previous designs is that: 1) we optimize and evaluate both convolution and deconvolution layers in our CNN accelerator, to support segmentation task; 2) we propose parametrized deconvolution accelerator to support different CNN architectures; 3) we focus on real-time response of image segmentation, and propose new net architecture to achieve lower latency; we also propose a non-linear optimization model for design space exploration, and automatically generate the optimal CNN design for any given network and FPGA device.

10 CONCLUSIONS

In this work, we propose deeply customized and parametrized deconvolution accelerator for CNN-based segmentation problem. We optimize the hardware architecture of the CNN accelerator for both convolution and deconvolution layer, by sharing the input buffer, design space exploration to achieve optimal parallelism parameters and data quantization. The parameter tuning is performed by solving the proposed non-linear optimization problem. Our current and future effort addresses the challenge of supporting 30 frames per second for 512*512 image segmentation, by exploring various enhancements of our designs based on techniques such as multi-pumping [31], and deploying to a larger device.

REFERENCES

[1] Vijay Badrinarayanan, Alex Kendall, and Roberto Cipolla. 2017. Segnet: A deep convolutional encoder-decoder architecture for image segmentation. *IEEE Trans. pattern analysis and machine intelligence.* 39, 12 (2017), 2481–2495.

[2] 2017. *Semantic Understanding of Urban Street Scenes: Benchmark Suite.* https://www.cityscapes-dataset.com/benchmarks/

[3] Marius Cordts, Mohamed Omran, Sebastian Ramos, Timo Rehfeld, Markus Enzweiler, Rodrigo Benenson, Uwe Franke, Stefan Roth, and Bernt Schiele. 2016. The cityscapes dataset for semantic urban scene understanding. In *CVPR*. 3213–3223.

[4] Vincent Dumoulin and Francesco Visin. 2016. A guide to convolution arithmetic for deep learning. arXiv:1603.07285. https://arxiv.org/abs/1603.07285

[5] Song Han, Junlong Kang, Huizi Mao, Yiming Hu, Xin Li, Yubin Li, Dongliang Xie, Hong Luo, Song Yao, Yu Wang, et al. 2017. ESE: Efficient Speech Recognition Engine with Sparse LSTM on FPGA. In *FPGA*. 75–84.

[6] Song Han, Huizi Mao, and William J Dally. 2015. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. (2015). arXiv:1510.00149. https://arxiv.org/abs/1510.00149

[7] Kaiming He and Jian Sun. 2015. Convolutional neural networks at constrained time cost. In *CVPR*. 5353–5360.

[8] Yihui He, Xiangyu Zhang, and Jian Sun. 2017. Channel pruning for accelerating very deep neural networks. In *ICCV*, Vol. 2. 6.

[9] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. Mobilenets: Efficient convolutional neural networks for mobile vision applications. (2017). arXiv:1704.04861. https://arxiv.org/abs/1704.04861

[10] Phillip Isola, Jun-Yan Zhu, Tinghui Zhou, and Alexei A Efros. 2016. Image-to-image translation with conditional adversarial networks. (2016). arXiv:1611.07004. http://arxiv.org/abs/1611.07004

[11] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. 2014. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia*. ACM, 675–678.

[12] Shuanglong Liu and Christos-Savvas Bouganis. 2017. Communication-Aware MCMC Method for Big Data Applications on FPGAs. In *FCCM*. 9–16.

[13] Shuanglong Liu, Grigorios Mingas, and Christos-Savvas Bouganis. 2017. An unbiased mcmc fpga-based accelerator in the land of custom precision arithmetic. *IEEE Trans. Comput.* 66, 5 (2017), 745–758.

[14] Jonathan Long, Evan Shelhamer, and Trevor Darrell. 2015. Fully convolutional networks for semantic segmentation. In *CVPR*. 3431–3440.

[15] Liqiang Lu, Yun Liang, Qingcheng Xiao, and Shengen Yan. 2017. Evaluating fast algorithms for convolutional neural networks on fpgas. In *FCCM*. 101–108.

[16] Hyeonwoo Noh, Seunghoon Hong, and Bohyung Han. 2015. Learning deconvolution network for semantic segmentation. In *Proceedings of the IEEE International Conference on Computer Vision*. 1520–1528.

[17] Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Brucek Khailany, Joel Emer, Stephen W Keckler, and William J Dally. 2017. SCNN: An Accelerator for Compressed-sparse Convolutional Neural Networks. In *ISCA*. 27–40.

[18] Jiantao Qiu, Jie Wang, Song Yao, Kaiyuan Guo, Boxun Li, Erjin Zhou, Jincheng Yu, Tianqi Tang, Ningyi Xu, Sen Song, et al. 2016. Going deeper with embedded fpga platform for convolutional neural network. In *FPGA*. 26–35.

[19] Alec Radford, Luke Metz, and Soumith Chintala. 2015. Unsupervised representation learning with deep convolutional generative adversarial networks. (2015). arXiv:1511.06434. https://arxiv.org/abs/1511.06434

[20] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. 2015. U-net: Convolutional networks for biomedical image segmentation. In *In Medical Image Computing and Computer-Assisted Intervention*. Springer, 234–241.

[21] Ruslan Salakhutdinov. 2015. Learning deep generative models. *Annual Review of Statistics and Its Application* 2 (2015), 361–385.

[22] Wenzhe Shi, Jose Caballero, Ferenc Huszár, Johannes Totz, Andrew P Aitken, Rob Bishop, Daniel Rueckert, and Zehan Wang. 2016. Real-time single image and video super-resolution using an efficient sub-pixel convolutional neural network. In *CVPR*. 1874–1883.

[23] Amir Yazdanbakhsh, Michael Brzozowski, Behnam Khaleghi, Soroush Ghodrati, Kambiz Samadi, Hadi Esmaeilzadeh, and Nam Sung Kim. 2018. FlexiGAN: An End-to-End Solution for FPGA Acceleration of Generative Adversarial Networks. In *FCCM*.

[24] Amir Yazdanbakhsh, Kambiz Samadi, Hadi Esmaeilzadeh, and Nam Sung Kim. 2018. GANAX: A Unified SIMD-MIMD Acceleration for Generative Adversarial Network. In *ISCA*.

[25] Matthew D Zeiler, Dilip Krishnan, Graham W Taylor, and Rob Fergus. 2010. Deconvolutional networks. In *CVPR*. 2528–2535.

[26] Matthew D Zeiler, Graham W Taylor, and Rob Fergus. 2011. Adaptive deconvolutional networks for mid and high level feature learning. In *ICCV*. 2018–2025.

[27] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. 2015. Optimizing fpga-based accelerator design for deep convolutional neural networks. In *FPGA*. 161–170.

[28] Kai Zhang, Wangmeng Zuo, Yunjin Chen, Deyu Meng, and Lei Zhang. 2017. Beyond a gaussian denoiser: Residual learning of deep cnn for image denoising. *IEEE Transactions on Image Processing* 26, 7 (2017), 3142–3155.

[29] Xinyu Zhang, Srinjoy Das, Ojash Neopane, and Ken Kreutz-Delgado. 2017. A Design Methodology for Efficient Implementation of Deconvolutional Neural Networks on an FPGA. (2017). arXiv:1705.02583. http://arxiv.org/abs/1705.02583

[30] Ruizhe Zhao, Xinyu Niu, Yajie Wu, Wayne Luk, and Qiang Liu. 2017. Optimizing CNN-Based Object Detection Algorithms on Embedded FPGA Platforms. In *ARC*. Springer, 255–267.

[31] Ruizhe Zhao, Tim Todman, Wayne Luk, and Xinyu Niu. 2017. DeepPump: Multi-pumping deep neural networks. In *ASAP*. 206–206.

Received December 2017; revised April 2018; accepted July 2018