A Low-Latency Inference of Randomly Wired Convolutional Neural Networks on an FPGA

Ryosuke KURAMOCHI\(^{[a]}\), Nonmember and Hiroki NAKAHARA\(^{[b]}\), Member

SUMMARY Convolutional neural networks (CNNs) are widely used for image processing tasks in both embedded systems and data centers. In data centers, high accuracy and low latency are desired for various tasks such as image processing of streaming videos. We propose an FPGA-based low-latency CNN inference for randomly wired convolutional neural networks (RWCNNs), whose layer structures are based on random graph models. Because RWCNNs have several convolution layers that have no direct dependencies between them, our architecture can process them efficiently using a pipeline method. We build a conflict graph using the scheduling results. Then, we allocate the calculation results of each layer to the HBM2 channels by coloring the graph. The pipeline execution needs to be properly controlled, we developed an automatic generation tool for hardware functions. We implemented the proposed architecture on the Alveo U50 FPGA. We investigated a trade-off between latency and recognition accuracy for the ImageNet classification task by comparing the inference performances for different input image sizes. We compared our accelerator with a conventional accelerator for ResNet-50. The results show that our accelerator reduces the latency by 2.21 times. We also obtained 12.6 and 4.93 times better efficiency than CPU and GPU, respectively. Thus, our accelerator for RWCNNs is suitable for low-latency inference.

key words: randomly wired convolutional neural network, inference accelerator, FPGA, image recognition

1. Introduction

1.1 Low-Latency Computations for AI Applications

Artificial intelligence (AI) applications that use deep learning are critical for enabling a new era in which machines enhance human creativity. In particular, convolutional neural networks (CNNs) are widely used for computer-vision applications, such as segmentation \([1]\), object detection \([2]\)–\([4]\), and pose estimation \([5]\). The growing interest in using FPGAs to accelerate CNNs provides the driving force behind the deployment of FPGAs in cloud services, such as Amazon AWS and Microsoft Azure. The availability and flexibility of FPGAs in the cloud raise new challenges in the design and implementation of deep learning models on these platforms. The recent adoption of FPGA demonstrates its great ability to run CNN-related applications in both of the above cloud servers. The combination of programmable hardware and CNNs has resulted in many possibilities for reshaping the landscape of deep learning applications to achieve low latency (short-time response) and high energy efficiency.

Furthermore, in use cases where safety and human life are at risk, such as in driving-assistance systems and self-driving cars, a lower latency is extremely important, as system size and power constraints are more restricted than those in data centers. Tesla Inc. requested an AI chip for autonomous driving with an energy consumption of less than 100 W and low latency (batch size 1)\([6]\). The speed at which humans respond to visual input is within approximately a quarter of the second. However, automated driving systems ideally require quicker responses than humans. Actions such as emergency braking or detecting an object should be done in less than 1.5 seconds to ensure that an autonomous driving system is comparable to a human driver.

1.2 Randomly Wired Convolutional Neural Networks (RWCNNs)

A typical feed-forward CNN is illustrated in Fig. 1. We have to process the layers in order from front to back. Because it is difficult to process multiple layers in parallel, we cannot process them efficiently using a pipeline method. It is true that we can remove dependencies of successive layers by using a batch of two or more sizes, but this may increase latency and it is not suitable for inference tasks. In this work, we focus on randomly wired convolutional neural networks (RWCNNs)\([7]\), whose network structure is built based on random graphs. As depicted in Fig. 2, because

---

\(\text{Manuscript received January 8, 2021.}\\ 
\text{Manuscript revised April 23, 2021.}\\ 
\text{Manuscript publicized June 24, 2021.}\\ 
\text{The authors are with the Tokyo Institute of Technology,}\\ 
\text{Tokyo, 152–8550 Japan.}\\ 
\text{a) E-mail: kuramochi@reconf.ie.titech.ac.jp}\\ 
\text{b) E-mail: nakahara@ict.e.titech.ac.jp}\\ 
\text{DOI: 10.1587/transinf.2021PAP0010}\\

---
RWCNNs have many layers that can be processed in parallel, we can process them efficiently using a pipeline method by properly scheduling their processing order.

1.3 RWCNN Realization on an FPGA

AI accelerators based on FPGAs have been proposed to solve issues related to the performance, power consumption, and future adaptability of AI applications. FPGAs can be configured with hundreds or thousands of highly parallelized arithmetic units with a minimum of resolution of 1-bit and customized memory interfaces to eliminate their bottlenecks. Because FPGAs can be easily re-programmed, we can take full advantage of the performance of new networks such as an RWCNN. In particular, flexibility is the most important factor because the connections of layers in an RWCNN change depending on its configuration.

This work is an extension of our previous publication [8]. An overview of the changes is depicted in Fig. 3. In the previous work, we proposed a CNN inference accelerator for RWCNNs on an FPGA. To realize low latency inference, we took advantage of the parallelism of convolutional layers in random graphs and used two convolution engines (CEs) to perform convolutions in parallel. To efficiently execute RWCNNs, we scheduled orders of execution of convolutional layers and allocated each layer to HBM2 channels. Thus, we achieved lower latency than a conventional FPGA-based CNN accelerator. However, we faced a problem that as we increase the number of the CEs, utilization efficiency of them becomes low because of the limitation of the layer parallelism. This made it difficult to further improve the performance of our previously proposed accelerator. In this work, we take advantage of the layer parallelism differently. We use only one CE for convolutional layers in random graphs and it performs convolutions with an efficient pipeline method. We changed our architecture to support the pipeline execution and developed an automatic hardware generation tool for the pipeline. To show the superiority of our RWCNN accelerator, we conducted some additional experiments.

Our contributions are as follows:

1. We propose an FPGA-based CNN inference accelerator for RWCNNs. The accelerator performs convolutional operations efficiently using a pipeline method over feature map tiles and layers. Because RWCNNs have several layers that have no direct dependencies between them, we can remove almost all pipeline stalls and improve its efficiency by appropriately scheduling the layers.

2. Because the proposed architecture requires proper control for pipeline processing, we developed an automatic generation tool for hardware functions. This tool takes RWCNN graph information as an input and performs node scheduling, HBM2 channel allocation, and code generation of hardware functions for pipeline control.

3. We evaluated the latency and accuracy of the ImageNet classification task [9] for different input image sizes. The result shows that we can further lower the latency by approximately 19% at the cost of accuracy degradation of 0.8%.

4. We compared the proposed accelerator with other platforms. The results show that the proposed accelerator for RWCNNs is suitable for low-latency inference.

The rest of the paper is organized as follows: Sect. 2 introduces related works. Section 3 explains the basics of CNNs and Sect. 4 explains the RWCNNs used in this study. Section 5 describes an architecture to efficiently perform convolutional operations using a pipeline method. In Sect. 6, we describe node scheduling, HBM2 channel allocation, and an automatic hardware generation tool. Section 7 presents the experimental results and Sect. 8 concludes the paper.

2. Related Works

Several FPGA-based CNN accelerators have been proposed. Most of them lower the latency by using techniques such as increasing the parallelism for convolutional operations and reducing the required memory bandwidth. Reference [10] analyzed optimization techniques for CNN accelerators and proposed a high-performance CNN accelerator that maximized the efficiency of resource usage and minimized the amount of data transfer. In [11], a low-latency CNN accelerator was realized by processing CNNs in the spectral domain and applying the model-pruning technique to reduce convolutional operations.

In addition, quantization techniques are often used. The performance of inference tasks can be improved by using low bit precision data types in place of 32-bit floating-point precision. Although most studies [10]–[12] proposed CNN accelerators that take advantage of 8-bit or 16-bit fixed-point precision, there are some works [13], [14] which focus on inference tasks with extremely low bit widths, such as 1-bit or 2-bit.

In addition, there are CNN accelerators that improve their performance by targeting efficient CNN models. Compared with general CNN models such as AlexNet [15], VGG16 [16], GoogleNet [17] and ResNet [18], efficient models such as MobileNet [19], [20] and ShuffleNet [21] can achieve similar accuracy with fewer parameters and operations. Many studies [22]–[24] have proposed CNN inference accelerators for MobileNetV1, which has depthwise separable convolution [25] as used in this research. Furthermore, inference accelerators targeting MobileNetV2, which

![Fig. 3](image-url)
performs convolutional operations with inverted residual blocks have also been proposed [26]–[28]. Reference [29] proposed a new CNN model that was suitable for hardware execution and realized low-latency inference on an FPGA. As in these works, we propose an FPGA-based CNN inference accelerator that targets the specific CNN model, RWCNN. Our accelerator can perform inference with an efficient pipeline method by taking advantage of its layer parallelism.

3. Definition

A typical CNN consists of a group of $L$ sequential layers, including convolutional, pooling, and fully-connected (FC) layers. We assume that a pretrained model is available, and the goal is to realize only inferences with low latency on an FPGA. In the following section, we present conventional techniques used in the study.

3.1 Convolutional Operation

Figure 4 shows the convolutional operation. The forward two-dimensional (2D) convolution layer computes output $Y$ by applying a set of $m$ 2D kernels $k \times k$ to $n$ input feature maps $X$. The following formula is applied:

$$ y_{ij}^{(m)} = \sum_{n} y_{ij}^{(m,n)} = \sum_{n} \sum_{s=0}^{k-1} \sum_{t=0}^{k-1} w_{s,t}^{(m,n)} x_{i+s,j+t} + b^{(m)}. $$

3.2 Depthwise Separable Convolution

Conventional convolutional operations perform in both spatial and channel directions of the input feature maps at a time, whereas depthwise separable convolution performs a convolution independently in the spatial and channel directions. It is based on the hypothesis that convolutions can be separated in these directions. The depthwise separable convolution is shown in Fig. 5. Spatial convolution and a channel direction convolution are also referred to as depthwise convolution (Dwcv) and pointwise convolution (Pwcv), respectively. Dwcv is a process of independently performing convolution in the spatial direction independently for each channel of the feature maps, while Pwcv is a $1 \times 1$ convolution operation. Consider $n \times r \times c$ input feature maps size and $m$ output feature maps. For $k \times k$ convolution, its computation order becomes $O(nrc^2m)$. The computation orders for a depthwise and a pointwise convolution are $O(nrc^2)$ and $O(nrcm)$, respectively. By converting the conventional $k \times k$ convolution to a separable convolution (a pair of depthwise and pointwise convolutions), its computation order can be relaxed to $O(nrc^2 + nrcm)$. Typically, $m \gg k^2$ (e.g., $k = 3$ and $m = 64$); thus, computations of the separable convolution can be reduced by a factor of $1/k^2$.

4. Randomly Wired Convolutional Neural Networks

Randomly wired convolutional neural networks (RWCNNs) were originally proposed in [7]. We will briefly explain how to compose one.

4.1 Generating Graphs

A network generator creates a set of nodes and edges that connect nodes. Once a graph is obtained, it is mapped to a computable neural network using a simple mapping method described as follows.

4.1.1 Edge Operations

Assuming that the graph is directed by construction, we define the edges as data flow; that is, a directed edge sends data (a tensor) from one node to another one.

4.1.2 Node Operations

A node in a directed graph may have some input edges and some output edges. As shown in Fig. 2, we define the operations represented by one node as follows:

- Aggregation: The input data (from one or more edges) to a node are combined using a weighted sum.
- Transformation: The aggregated data are processed via a transformation defined as a ReLU-convolution-batch normalization [30] triplet [31]. A $3 \times 3$ depthwise separable convolution [25] is used for all the nodes.
- Distribution: The same copy of the transformed data is sent out by the output edges of the node.
Table 1  RWCNN model structure used in this study. GAP denotes a global average pooling layer [32].

| In. F.Size | Stage        | #Node. | #In. | #Out. |
|------------|--------------|--------|------|-------|
| 224x224    | 3x3 Conv     | -      | 3    | 54    |
| 112x112    | Random       | 16     | 54   | 109   |
| 56x56      | Random       | 32     | 109  | 218   |
| 28x28      | Random       | 32     | 218  | 436   |
| 14x14      | Random       | 32     | 436  | 872   |
| 7x7        | 1x1 Conv     | -      | 872  | 1280  |
| 1x1        | 1x1 Conv     | -      | 1280 | 1280  |

Table 2  Comparison of ResNet-50 and RWCNN reported in [7]. ImageNet2012 1000-class classification task [9] is used for the accuracy comparison.

| network    | FLOPs (×10^9) | params (×10^6) | Accuracy (%) |
|------------|---------------|----------------|--------------|
| ResNet-50  | 4.1           | 25.6           | 77.1         |
| RWCNN      | 4.0           | 51.9           | 79.0         |

Parameter $P$, and it is denoted by ER($P$).

4.5 Advantages of RWCNN

A comparison between RWCNN and ResNet-50 [18] is presented in Table 2. Both models require almost the same amount of computation. Although we need more parameter transfers for RWCNNs, we can obtain a 1.9% higher ImageNet top-1 recognition accuracy. For hardware execution of RWCNNs, our proposed architecture can perform inference tasks with an efficient pipeline method by properly scheduling the order of node execution. The hardware implementations are explained in the following sections.

5. FPGA Implementation

5.1 Overview

Because most of the layers of RWCNN are depthwise separable convolution [25], $k \times k$ convolution and global average pooling [32] layers, we accelerate these layers by using an FPGA. An overview of our proposed architecture is depicted in Fig. 7. We use a separable convolution engine (SCE), an initial convolution engine (ICE) and a global average pooling engine (GAPE) to process a depthwise separable convolution, a convolution at the first layer, and a global average pooling, respectively. The SCE needs a weighted sum of calculation results of multiple nodes as input. Therefore, we use HBM2 to store the feature maps, thereby enabling simultaneous memory access with multiple channels. We connect the SCE and an aggregation engine (AE) to realize the weighted sum.

When processing a CNN, we store the weight parameters of each RWCNN node and input image values in the HBM2 by using PCI Express (PCIe) in advance. Prior to processing nodes at the SCE, the required input feature maps are read from the HBM2 channels via the AE. After the SCE completes the processing, the results are stored back to the HBM2 channels allocated to each node. After the calculations of all the layers are complete, the results are returned to a HOST CPU via PCIe.

5.2 Convolution Engine

SCE processes the nodes in turn. It performs a depthwise separable convolution, which consists of a depthwise convolution (Dwcv) and a pointwise convolution (Pwcv). Because of the limited number of on-chip memories on an FPGA and different sizes of feature maps between the stages of an RWCNN, we use a tiling technique to perform convolutional
operations, as performed in [10] and [35]. As depicted in Fig. 8, rows and columns of output feature maps are divided by \( T_R \) and \( T_C \), respectively, channels of input feature maps by \( T_N \), and weight filters by \( T_M \). Assuming that \( \text{PAD} \) denotes the number of zero paddings, rows and columns of input feature maps are divided by \( T_R + 2\text{PAD} \) and \( T_C + 2\text{PAD} \), respectively.

A convolution unit (CU) array is depicted in Fig. 9. In the ICE and SCE, we realize the convolutions by using the CU array. The CU has local memory to store weight parameters (W.mem) and several processing elements (PEs). At each PE, we use feature map data and weight parameters to calculate the inner product in a channel direction. Each CU can compute several output neurons in parallel by having \( P_R \) and \( P_C \) PEs in row and column directions of the feature maps, respectively. We also use \( P_M \) CUs in an output channel direction. Thus, we can compute a total of \( P_M \times P_R \times P_C \) number of output neurons in parallel.

The architecture of the PE is shown in Fig. 10. The PE requires an input feature maps vector of length \( P_N \) in a channel direction and a weight vector of the same length. We realize an inner product of vectors by computing element-wise products and then adding them up with an adder tree. The calculated result is stored to a temporary register and released after all calculations are complete. Especially for Dwcv, we set the \( P_N \) to 1 and remove the adder tree because convolutional operations along channel direction are not necessary. As the PE computes \( P_N \) elements at each cycle, the CU array can perform convolutional operations with a parallelism of \( P_M \times P_N \times P_R \times P_C \).

Different convolutional operations can be realized by setting the appropriate parameters and data flows for the CU array. For Conv and Pwcv, we broadcast the same input neurons to the CU array from the local memory (FM.mem) because the same input neurons are used at each CU. For Dwcv, each CU loads input neurons corresponding to each channel. In addition, we broadcast the same weight parameters from each W.mem because the same weight parameters are used at the PEs in each CU. By reusing feature map data and weight parameters in this way, we can efficiently perform convolutional operations. We perform convolutional operations along row and column directions by providing a different weight parameter and a corresponding input neuron at each cycle. Thus, we need \( K \times K \) cycles to complete a \( K \times K \) convolution.

Figures 11 and 12 show the pipeline executions of convolutions and depthwise separable convolutions, respectively. We perform the convolutions with the 3-stage pipeline: reading input feature maps (Read FM), applying the convolutional operation (Conv), and writing output feature maps (Write FM). We also perform the depthwise separable convolutions with the 4-stage pipeline: Read FM, Dwcv, Pwcv, and Write FM. In addition, we need to insert a pipeline stage of the reading weight parameters (Read W) at every start of the tile processes. Modules that perform each process are connected using double buffering; therefore, the processes can be performed in parallel.

In particular, for the pipeline execution of the depthwise separable convolution, it is important to set the degree of parallelism so that the execution times of Pwcv and Dwcv are as equal as possible. The number of operations for Pwcv is denoted by \( C_{\text{pwc}} \) and that for Dwcv by \( C_{\text{dwc}} \). They are represented as follows:

\[
C_{\text{pwc}} = \frac{m \cdot n \cdot r \cdot c}{P_{M,\text{pwc}} \cdot P_{N,\text{pwc}} \cdot P_{R,\text{pwc}} \cdot P_{C,\text{pwc}}}
\]

\[
C_{\text{dwc}} = \frac{n \cdot r \cdot c \cdot k^2}{P_{M,\text{dwc}} \cdot P_{C,\text{dwc}} \cdot P_{R,\text{dwc}}}
\]
If we set \( P_{N,\text{pucv}} \) and \( P_{M,\text{dwcv}} \) to be equal in order to keep the buffers between them simple, ratio \( R \) of the computational complexity of \( P_{\text{c},wcv} \) to that of \( D_{\text{c},wcv} \) is calculated as follows:

\[
R = \frac{C_{\text{pucv}}}{C_{\text{dwcv}}} = \frac{m \cdot P_{R,\text{dwcv}} \cdot P_{C,\text{dwcv}}}{k^2 \cdot P_{M,\text{pucv}} \cdot P_{R,\text{pucv}} \cdot P_{C,\text{pucv}}}.
\]

We adjust the values of \( P_{R,\text{dwcv}}, P_{C,\text{dwcv}}, P_{M,\text{pucv}}, P_{R,\text{pucv}} \), and \( P_{C,\text{pucv}} \) so that \( R \) is as close as possible to 1 for all the stages.

5.3 Aggregation Engine (AE)

When multiple edges are connected to a node, their weighted sum is used for the input values of the node to realize aggregation. However, if we perform weighted addition every time an input node is processed, multiple accesses to the off-chip memory will result in poor performance. As shown in Fig. 13, we implement an architecture for aggregation. Prior to processing a node, several feature map values are read from HBM2 channels. A weighted sum of them is calculated by performing MAC operations of the read values and the weights of the edges. This reduces the instances of access of each node to the feature map memory, resulting in the efficient processing of an RWCNN. We describe a method for allocating nodes to HBM2 channels in Sect. 6.2. We realize the aggregation by applying a ReLU function to the weighted sum at the end.

An extra node connected to the first nodes of each stage does not perform any convolutions; it only distributes the results from its previous stage to them. Similarly, an extra node connected to the last nodes only aggregates their results. To efficiently process an RWCNN, we remove the extra nodes and directly transfer the values from the output nodes of a stage to the input nodes of the next stage.

6. Design Method

6.1 Node Scheduling

As shown in Sect. 5.2, our proposed architecture realizes the depthwise separable convolutions using the pipeline method. Therefore, it is important to prevent pipeline stalls to improve efficiency. In general CNN models, there is a direct dependency between two consecutive layers as shown in Fig. 14. Because we have to process the layers in order, it is difficult to overlap the executions of the layers. Even by splitting convolutions with the tiling technique, it is still difficult to overlap because consecutive layers need to access to the same off-chip memory bank. This causes pipeline stalls and leads to poor performance. By increasing the batch size, we can remove direct dependencies and prevent pipeline stalls. However, it will cause high latency and will be inappropriate for inference tasks. In contrast, there are fewer direct dependencies between RWCNN nodes as depicted in Fig. 15. Therefore, by scheduling the processing order of nodes to avoid dependencies as much as possible, we can decrease pipeline stalls and realize efficient processing of RWCNNs.

Because the dependencies between RWCNN nodes do not change during inference, we determine the execution order before inference. We prioritize nodes that have no dependencies on a previously allocated node and belong to the longest path. In this way, we schedule nodes to minimize pipeline stalls.
6.2 Node Allocation to the HBM2 Channels

To optimally use the HBM2 channels, we need to allocate the calculation results of nodes to them. If we allocate nodes and their input nodes to the same HBM2 channels, then the HBM2 channel reads and writes will have a conflict between them, resulting in significant performance degradation. To prevent this, nodes must be allocated to the HBM2 channels so that the calculation results of those nodes are stored in different HBM2 channels.

We allocate nodes to the HBM2 channels by considering the coloring problem of a conflict graph that has edges between nodes that are simultaneously read or written. We use the Welsh-Powell algorithm [36] to solve this problem. Therefore, nodes can be allocated to HBM2 channels corresponding to their colors. We use some of the HBM2 channels that are not used for node allocation to transfer weight parameters to the CEs.

6.3 Automatic Generation of Hardware Functions

The dataflow of the RWCNN is complicated, and it is necessary to control it properly. Although we can control it by sequentially sending instructions from the HOST CPU, interactions between the HOST CPU and the accelerator will affect the overall performance. Therefore, we developed an automatic generation tool for the hardware functions for the pipeline control. As shown in Fig. 16, we first input information about the connections between the nodes. Next, the tool performs node scheduling and HBM2 channel allocation. Finally, it generates hardware functions that can properly control the pipeline execution. The generated hardware functions are converted to RTL by using high-level synthesis (HLS).

7. Experimental Results

7.1 Implementation Results

Because parallel access to the off-chip memory is essential for efficient processing of the RWCNN, we implemented our proposed architecture on the Xilinx Alveo U50 Data Center Accelerator Card (XCU50, 2,688 18Kb BRAMs, 5,952 DSP48E2 blocks, 1,743,360 FFs, 871,680 LUTs, 640 288Kb URAMs, and two 4GB HBM2) with a constraint of 125 MHz. We used the Xilinx Vitis 2020.1 and host PC with Ubuntu 16.04 OS, Intel Core i7-8700K CPU, and DDR4 64GBytes memory. The tiling parameters and the parallelism parameters for Conv, Dwcv, and Pwcv are listed in Table 3. We used the PyTorch deep learning framework [37] version 1.3.1 to train and evaluate the RWCNN. Table 4 shows the hardware resource usage obtained after logic synthesis. Because we apply 8-bit quantization, we can perform double MAC operations with a DSP48E2 block [38]. We allocate a large amount of resources to the SCE because most of the processing of an RWCNN is depthwise separable convolutions. Table 5 shows the results of applying the node scheduling and the channel allocation to the RWCNN model used in this experiment.

7.2 Comparison between Different Image Sizes

Table 6 shows a comparison of the recognition accuracy and latency for different input image sizes. We used quantization aware training [39] for 8-bit fixed-point training. Because we need to halve the size of feature maps five times in total, we use image sizes that are multiples of $2^5$ for the comparison. Because our proposed architecture performs convolutional operations with the tiling technique, the amount of resources is almost equal for all the input image sizes used in this comparison. Therefore, the degree of parallelism does not change and only the number of operations required for inference varies in proportion to the image size. From this result, there is a trade-off between the recognition accuracy and the latency; therefore, it is important to select an appropriate input image size according to applications. Comparing the results of $224 \times 224$ and that of $192 \times 192$, we can see that the latency can be reduced by approximately 19%
at the cost of only 0.8% accuracy degradation. Figure 17 shows the ratio of the amount of data transferred between the FPGA and the HBM2. This result indicates that as the image size decreases, the ratio of time to transfer weight parameters increases, which causes a decrease in throughput. Therefore, by using more HBM2 channels for weight parameter transfer, we can improve the scalability with respect to image size.

7.3 Comparison with an FPGA-Based Accelerator

Table 7 presents a comparison of our accelerator with an FPGA-based accelerator for ResNet-50 [12]. Although RWCNN has a computational complexity comparable to that of ResNet-50, it can achieve about 1.0% higher top-1 recognition accuracy in the ImageNet image-classification task. The comparative result shows that our accelerator that targets RWCNN can process an image in 13.1 ms, which is approximately 55% lower than the result of [12].

7.4 Comparison between Different Platforms

To show the advantage of using an FPGA, we compare the latency of the RWCNN executed on a CPU, a GPU, and an FPGA. To compare the lowest achievable latency on each device, we set the batch size to 1. We implemented a software program with the PyTorch deep learning framework and used it for the inference on the CPU and GPU. For the CPU inference, we used an Intel Core i7-8700K CPU and applied INT8 quantization. For GPU inference, we used a NVIDIA Quadro GV100 GPU, which equipped with 32GB HBM2, and we applied FP16 quantization. In this experiment, we used a larger size model whose channel size of convolution layers is a power of 2. We found that without this modification, the latency of the INT8-quantized model becomes much higher than that of the FP32 model. The throughput result of the FPGA becomes higher than the result in Sect. 7.3. This is because the utilization rate of DSP blocks becomes higher by modifying the channel sizes.

Table 8 shows the comparison results. Our accelerator achieves 12.6 times higher efficiency than the CPU, and 4.93 times than the GPU. The CPU can perform convolutional operations efficiently with SIMD instructions. However, the latency is higher than the others probably because its degree of parallelism is lower. It also suffers from high power consumption due to the overhead of circuits for increasing its instruction-level parallelism. We expect that although higher performance CPUs can complete inference with lower latency, they will suffer from higher power consumption and significant improvement in efficiency cannot be achieved. The GPU can also perform convolutional operations efficiently with SIMD processors. However, with the limited batch size, the utilization rate of the processors will be low and this will cause high latency. We can assume that the throughput result of the GPU will be the highest for a larger batch size. In contrast, our accelerator utilizes not only data-level parallelism with the CU arrays but also layer parallelism with the efficient pipeline method. This improves the utilization rate of DSP blocks during inference, resulting in lower latency.

7.5 Comparison with the Previously Proposed Accelerator

Table 9 shows the comparison with our previous work [8]. The proposed accelerator in this work can achieve lower latency with a lower operating frequency. This indicates that with the pipeline method, the proposed accelerator can perform convolutional operations more efficiently. The
Table 9 Performance and resource usage comparisons with the previous accelerator[8].

| Design | [8]      | This work |
|--------|----------|-----------|
| CNN model | RWCNN  | RWCNN  |
| FPGA   | Alveo U50 | Alveo U50 |
| Precision [bit] | 8   | 8   |
| Clock [MHz]    | 200  | 125  |
| LUT (%)        | 43.39 | 29.08 |
| DSP (%)        | 78.09 | 60.95 |
| BRAM (%)       | 46.39 | 60.31 |
| URAM (%)       | 57.50 | 70.00 |
| Latency [ms]   | 16.6  | 13.1  |
| Throughput [GOPS] | 474.7 | 601.5 |

previous accelerator processed convolutional layers in random graphs by using several independent CEs. This design method prevented the transfer of intermediate calculation results between CEs, and we assume that it led to a lower routing delay, resulting in a higher operating frequency. In contrast, since the proposed accelerator in this work use only one large CE for layers in random graphs, there are several routings for long distances inside the CE. Thus, we suffer from timing violations during logic placement and routing and achieved lower operating frequency. We assume that it can be improved by incorporating more efficient computational architectures like the systolic array architecture. As for resource usage, utilization ratios of the LUTs and DSPs become lower while those of the BRAMs and URAMs become higher. Since we reduce the number of CEs to one, the overheads of convolution circuits are also reduced, resulting in lower usages of the LUTs and DSPs. On the other hand, we use more on-chip RAMs in this work because the proposed accelerator uses the ping-pong buffers for the pipeline execution while the previous accelerator mainly used stream interconnections.

8. Conclusion

We proposed a low-latency CNN inference engine for RWCNNs. Because the network structure of RWCNNs is built based on random graph models, they have many layers that have no direct dependencies between them. Therefore, by appropriately scheduling them, our proposed architecture can perform convolutional operations with an efficient pipeline. For the hardware implementation, we used an FPGA with HBM2 because we needed parallel access to the computational results of previous layers. An automatic hardware generator was developed to realize the pipeline. It takes graph information as an input and generates hardware functions after performing node scheduling and HBM2 channel allocation. We implemented our proposed accelerator on the Xilinx Alveo U50 Data Center Accelerator Card. We researched the trade-off between recognition accuracy and latency for different input image sizes. For the ImageNet2012 classification task, we showed that the latency could be reduced by 19% at the expense of 0.8% accuracy degradation. We compared the results with a conventional implementation for ResNet-50. The results show that our accelerator can achieve a 2.21 times lower latency than the existing FPGA implementation. We also compared our FPGA-based implementation with a CPU-based and a GPU-based implementation, and achieved 12.6 and 4.93 times higher efficiency, respectively. Therefore, our accelerator for RWCNNs is suitable for low-latency applications.

Acknowledgments

This research is supported in part by the Grants in Aid for Scientific Research of JSPS, Industry-academia collaborative R&D programs centre of innovation (COI) program, Core Research for Evolutional Science and Technology (CREST), and the New Energy and Industrial Technology Development Organization (NEDO). Also, thanks to the Xilinx University Program (XUP), Intel University Program, and the NVIDIA Corp.’s support.

References

[1] V. Badrinarayanan, A. Kendall, and R. Cipolla, “Segnet: A deep convolutional encoder-decoder architecture for image segmentation,” IEEE Trans. Pattern Anal. Mach. Intell., vol.39, no.12, pp.2481–2495, Dec. 2017.
[2] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S.E. Reed, C.Y. Fu, and A.C. Berg, “SSD: Single shot multibox detector,” Proc. European Conf. Comput. Vis. (ECCV), vol.9905, pp.21–37, 2016.
[3] J. Redmon and A. Farhadi. “Yolo9000: Better, faster, stronger,” 2017 IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR), pp.6517–6525, 2017.
[4] S. Ren, K. He, R. Girshick, and J. Sun, “Faster R-CNN: Towards real-time object detection with region proposal networks,” IEEE Trans. Pattern Anal. Mach. Intell., vol.39, no.6, pp.1137–1149, June 2017.
[5] Z. Cao, T. Simon, S. Wei, and Y. Sheikh, “Realtime multi-person 2d pose estimation using part affinity fields,” CVPR, pp.1302–1310, 2017.
[6] https://ir.tesla.com/events/event-details/tesla-autonomy-investor-day.
[7] S. Xie, A.Kirillov, R. Girshick, and K. He, “Exploring randomly wired neural networks for image recognition,” 2019 IEEE/CVF Int. Conf. Comput. Vis. (ICCV), pp.1284–1293, 2019.
[8] R. Kuramochi and H. Nakahara, “An FPGA-based low-latency accelerator for randomly wired neural networks,” 2020 30th Int. Conf. Field-Programmable Logic and Applications (FPL), pp.298–303, 2020.
[9] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A.C. Berg, and L. Fei-Fei, “ImageNet large scale visual recognition challenge,” Int. J. Comput. Vis. (IJCV), vol.115, no.3, pp.211–252, 2015.
[10] Y. Ma, Y. Cao, S. Vrudhula, and J. Seo, “Optimizing the convolution operation to accelerate deep neural networks on FPGA,” IEEE Trans. Very Large Scale Integr. (VLSI) Syst., vol.26, no.7, pp.1354–1367, July 2018.
[11] Y. Niu, R. Kannan, A. Srivastava, and V. Prasanna, “Reuse kernels or activations?: a flexible dataflow for low-latency spectral CNN acceleration,” Proc. 2020 ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays (FPGA), pp.266–276, Association for Computing Machinery, Feb. 2020.
[12] S. Hadjis and K. Olukotun, “Tensorflow to cloud FPGAs: Tradeoffs for accelerating deep neural networks,” FPL, pp.360–366, 2019.
[13] Y. Unuroglu, N.J. Fraser, G. Gambardella, M. Blott, P. Leong, M. Jahre, and K. Vissers, “Finn: A framework for fast, scalable binarized neural network inference,” FPGA, pp.65–74, Association for Computing Machinery, Feb. 2017.
[14] P. Colangelo, N. Nasiri, E. Nurvitadhi, A. Mishra, M. Margala, and K. Nealis, “Exploration of low numeric precision deep learning inference using intel® FPGAs,” 2018 IEEE 26th Annual Int. Symp. Field-Programmable Custom Computing Machines (FCCM), pp.73–80, 2018.

[15] A. Krizhevsky, I. Sutskever, and G.E. Hinton, “Imagenet classification with deep convolutional neural networks,” Advances in Neural Information Processing Systems 25, ed. F. Pereira, C.J.C. Burges, L. Bottou, and K.Q. Weinberger, vol.1, pp.1097–1105, Curran Associates, Inc., Dec. 2012.

[16] K. Simonyan and A. Zisserman, “Very deep convolutional networks for large-scale image recognition,” CoRR, vol.abs/1409.1556, 2015.

[17] C. Szegedy, Wei Liu, Yangqing Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, “Going deeper with convolutions,” CVPR, pp.1–9, June 2015.

[18] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” CVPR, pp.770–778, June 2016.

[19] A.G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, “Mobilenets: Efficient convolutional neural networks for mobile vision applications,” CoRR, vol.abs/1704.04861, 2017.

[20] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L. Chen, “Mobilenetv2: Inverted residuals and linear bottlenecks,” CVPR, pp.4510–4520, 2018.

[21] X. Zhang, X. Zhou, M. Lin, and J. Sun, “Shufflenet: An extremely efficient convolutional neural network for mobile devices,” CVPR, pp.6868–6876, 2018.

[22] J. Su, J. Faraone, J. Liu, Y. Zhao, D.B. Thomas, P.H.W. Leong, and P.Y.K. Cheung, “Redundancy-reduced mobilenet acceleration on reconfigurable logic for imagenet classification,” Applied Reconfigurable Computing. Architectures, Tools, and Applications, ed. N. Voros, M. Huebner, G. Keramidas, D. Goehringer, C. Antonopoulos, and P.C. Diniz, vol.10824, pp.16–28, Springer International Publishing, 2018.

[23] R. Zhao, X. Niu, and W. Luk, “Automatic optimizing CNN with depthwise separable convolution on FPGA: (abstract only),” FPGA, p.285, Association for Computing Machinery, Feb. 2018.

[24] B. Liu, D. Zou, L. Feng, S. Feng, P. Fu, and J. Li, “An FPGA-based CNN accelerator integrating depthwise separable convolution,” Electronics, vol.8, no.3, p.281, March 2019.

[25] F. Chollet, “Xception: Deep learning with depthwise separable convolutions,” CVPR, pp.1800–1807, 2017.

[26] L. Bai, Y. Zhao, and X. Huang, “A CNN accelerator on FPGA using depthwise separable convolution,” IEEE Trans. Circuits Syst. I, Exp. Briefs, vol.65, no.10, pp.1415–1419, Oct. 2018.

[27] D. Wu, Y. Zhang, X. Jia, L. Tian, T. Li, L. Sui, D. Xie, and Y. Shan, “A high-performance CNN processor based on FPGA for mobilenets,” FPL, pp.136–143, 2019.

[28] J. Knapehe, B. Stabernack, and M. Kuhnke, “A high throughput mobilenetv2 FPGA implementation based on a flexible architecture for depthwise separable convolution,” FPL, pp.277–283, 2020.

[29] Y. Yang, Q. Huang, B. Wu, T. Zhang, L. Ma, G. Gambardella, M. Blott, L. Lavagno, K. Vissers, J. Wawrzynek, and K. Keutzer, “Synety: Algorithm-hardware co-design for convnet accelerators on embedded FPGAs,” FPGA, p.23–32, Association for Computing Machinery, Feb. 2019.

[30] S. Ioffe and C. Szegedy, “Batch normalization: Accelerating deep network training by reducing internal covariate shift,” Proc. 32nd Int. Conf. Machine Learning, ed. F. Bach and D. Blei, Proc. Machine Learning Research, vol.37, pp.448–456, 07–09 July 2015.

[31] K. He, X. Zhang, S. Ren, and J. Sun, “Identity mappings in deep residual networks,” ArXiv, vol.abs/1603.05027, 2016.

[32] M. Lin, Q. Chen, and S. Yan, “Network in network,” CoRR, vol.abs/1312.4400, 2014.

[33] E.N. Gilbert, “Random graphs,” Ann. Math. Statist., vol.30, no.4, pp.1141–1144, Dec. 1959.

[34] P. Erdős and A. Rényi, “On the evolution of random graphs,” Publ. Math. Inst. Hungary, Acad. Sci., vol.5, pp.17–61, 1960.

[35] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, “Optimizing FPGA-based accelerator design for deep convolutional neural networks,” FPGA, pp.161–170, Feb. 2015.

[36] D.J.A. Welsh and M.B. Powell, “An upper bound for the chromatic number of a graph and its application to timetabling problems,” The Computer Journal, vol.10, no.1, pp.85–86, Jan. 1967.

[37] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintallat, “Pytorch: An imperative style, high-performance deep learning library,” Advances in Neural Information Processing Systems 32, ed. H. Wallach, H. Larochelle, A. Beygelzimer, F. d’Alché-Buc, E. Fox, and R. Garnett, pp.8024–8035, Curran Associates, Inc., 2019.

[38] Y. Fu, E. Wu, V. Santhaseelan, K. Denolf, K. Khan, and V. Kathail, “Embedded vision with int8 optimization on Xilinx devices,” Xilinx, San Jose, CA, USA, White Paper WP490 (v1.0.1), 2017.

[39] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, and D. Kalenichenko, “Quantization and training of neural networks for efficient integer-arithmetic-only inference,” CVPR, pp.2704–2713, 2018.

[40] H. Kung and C. Leiserson, “Systolic arrays (for VLSI),” in Sparse Matrix Proceedings, vol.1, pp.256–282, 1979.

[41] X. Wei, C.H. Yu, P. Zhang, Y. Chen, Y. Wang, H. Hu, Y. Liang, and J. Cong, “Automated systolic array architecture synthesis for high throughput CNN inference on FPGAs,” 2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC), pp.1–6, June 2017.