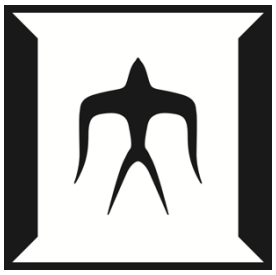
論文 / 著書情報
Article / Book Information

題目(和文)	FPGAベースの機械学習アクセラレータの設計最適化に関する研究
Title(English)	A Study on Design Optimization for FPGA-based Machine Learning Accelerator
著者(和文)	神宮司明良
Author(English)	Akira Jinguji
出典(和文)	学位:博士（工学）, 学位授与機関:東京工業大学, 報告番号:甲第11761号, 授与年月日:2022年3月26日, 学位の種別:課程博士, 審査員:中原 啓貴,髙橋 篤司,本村 真人,劉 載勲,佐々木 広,高前田 伸也
Citation(English)	Degree:Doctor (Engineering), Conferring organization: Tokyo Institute of Technology, Report number:甲第11761号, Conferred date:2022/3/26, Degree Type:Course doctor, Examiner:,,,,,
学位種別(和文)	博士論文
Type(English)	Doctoral Thesis

A Study on Design Optimization for FPGA-based Machine Learning Accelerator

Akira Jinguji



Tokyo Institute of Technology

March, 2022

Contents

Chapter 1

Preliminary

1.1 Background

A new paradigm of computer systems with machine learning is about to arrive, along with the practical application of the age of automated or unmanned societies. In the face of this new society, autonomous systems are approaching practical application, but they face several challenges. In general, a system consists of inputs, outputs, and processing. For instance, in an automated driving system, various sensors such as cameras are the input, and the gas pedal, brakes, and steering are the output, and the processing is to recognize the input information and determine how to drive. Existing technologies for input and output are sufficient for practical use, but recognition has been an issue. Until recently, recognition, an essential part of processing, has been a challenge, but with the development of machine learning, significant improvements in recognition performance have been realized, and the realization of autonomous systems is becoming realistic.

An autonomous system has several requirements: high recognition accuracy, real-time response time, low power consumption, and low manufacturing cost. There are several computing methods to achieve this, especially in embedded devices, the most typical of which is the CPU. CPUs in embedded devices are inexpensive and have low power consumption. However, their performance may be insufficient for real-time response. Therefore, the use of processors explicitly designed for the application, such as GPUs, ASICs, and FPGAs, can be considered. GPUs have high computational performance for matrix computation, but their power consumption is high, and they are not suitable for flexible computation, and designing dedicated circuits with ASICs requires a very high cost for design. We are interested in FPGAs because they are capable of very flexible computation and have a good balance

between moderately high-performance and relatively low design costs. FPGAs have a very high potential due to the miniaturization of semiconductor process size and hard macros. We also believe that by studying the excellent design of FPGAs, it is possible to exploit their high potential in terms of computing performance, power consumption, and design cost.

This research proposes an optimization design that satisfies all the constraints of computational performance, power consumption, and design cost on machine learning in autonomous systems. However, the following problem still exist; as for processing speed, an FPGA allows small and flexible circuit design, and its processing speed can be improved by co-designing with the machine learning algorithm itself; as for power consumption, there exists a trade-off between performance and power consumption; as for the design cost, it lacks a design flow for a low-cost device for machine learning. We are paying attention to FPGAs with high potential to meet all constraints with the above improvements. Our optimization method will bring us one step closer to realizing autonomous systems and accelerate the arrival of newcomers.

1.2 FPGA

1.2.1 Configuration

This section describes the structure of an FPGA. An FPGA consists of a grid of relatively small programmable logic blocks with vertical and horizontal channels between them. A single logic block is small, but a large circuit can be realized by combining many blocks. The basic structure of a programmable logic block is a look-up table (LUT) consisting of multiplexers and flip-flops. Flip-flops are used to construct synchronous outputs and sequential circuits. The flip-flop can be connected to the output of the LUT. The basic logic block is composed of this pair of LUT and flip-flop as the basic elements and hard macros to improve the performance. The structure of this block varies from manufacturer to manufacturer and is called CLB (configurable logic block), LE (logic element), or slice. The basic logic block can be arbitrarily connected through a switch matrix installed in the routing path. The switch matrix is made of transfer gates.

The LUTs that make up the logic of an FPGA are volatile memory, and the transfer gates that make up the switch matrix are also volatile memory. SRAM data is lost when the power is turned off. Therefore, the FPGA reads the circuit information (configuration data) from the outside when the power is turned on. In FPGAs, the delay time of the same logic varies depending

on placement and routing. Because FPGAs combine many small logic blocks and line resources, the length of the wiring and the number of switches to be routed through vary depending on where the logic blocks are actually placed. In the case of LSIs manufactured in a microprocessor, the delay of the placement wiring is larger than the delay of the logic elements. FPGAs are manufactured and provided as standard LSIs. However, they can also be used as custom LSIs because they are programmable, and their functions can be freely designed. In application-specific integrated circuits (ASICs), gate arrays and standard cells are often used. Since ASICs are manufactured in semiconductor factories, they require high development costs and a certain amount of time to manufacture. However, the price per unit can be reduced through mass production. On the other hand, FPGAs require more cost to manufacture the circuit, but the cost of writing the circuit data is negligible. The cost of writing circuit data is negligible, and the manufacturer can configure a single unit.

1.2.2 Architectural Features

GPUs use processor cores (CUDA cores) and a high bandwidth data transfer bus to execute threads in a workgroup based on a data-parallel model. On the other hand, FPGAs are designed based on a pipelined model because the data communication bandwidth to the CPU and the off-chip memory bandwidth is narrow. If there is no stall in the pipeline, the communication with the off-chip memory becomes the input/output of the pipeline. Thus, the operation can be performed efficiently, even in a narrow bandwidth. The on-chip memory of FPGAs is larger than that of GPUs and has a dual-port, so data transfer between pipeline stages can often be done with on-chip memory. FPGAs have another advantage because custom pipeline stages can be built in parallel. When the latency of each parallel operation differs, the GPU is slowed down by the slowest operation (warp divergence). However, the FPGA can construct a parallel architecture with uniform latency by allocating hardware resources appropriately. In addition to LUTs, FPGAs have high-density hard macros called DSPs and BRAMs. Unlike logic circuit design with LUTs, hard macros are implemented as ASIC circuits with specific functions and thus can realize high operating frequency operations and large memory capacity at high speed.

1.2.3 High-Level Synthesis (HLS) Design

The design of FPGA logic circuits using hardware description language (HDL) is much more time-consuming than the software-based design of CPUs and

GPUs. Therefore, in recent years, attention has been focused on high-level synthesis tools that synthesize logic circuits from the C programming language. High-level synthesis is creating a hardware description language based on a program written in a relatively high-level language to realize a logic circuit. In the conventional RTL level, it is necessary to design the specific operation at the clock cycle level, which makes it difficult to verify the operation. High-level synthesis allows design at a high level of abstraction using high-level languages suitable for algorithm description, thus improving design efficiency. Resource and clock constraints can be changed by configuration, and by inserting directives, pipelined, parallel, and standards-compliant input/output and control circuits can be easily generated. By inserting directives, pipelined, parallel, and standards-compliant input/output and control circuits can be easily generated. In addition, existing test tools can be used to verify the operation easily. Xilinx's Vivado HLS [1] and Intel's Intel FPGA SDK for OpenCL [2] have been put to practical use.

HLS design can be achieved in a relatively short period. For example, Yang et al.'s study [3], which proposes an FPGA implementation using HLS, was implemented by two workers in one month. On the other hand, there is a large difference in the achievable operating frequency between the HLS-generated design and the manual design using HDL. Because it is difficult to estimate the interconnect delay at the HLS level accurately. This problem is even more pronounced when implementing large HLS designs [4].

1.3 Machine Learning (ML)

1.3.1 ML for Embedded Systems

Machine learning is a method of analyzing data in which a machine learns laws from data and recognizes patterns behind the data. Machine learning applications cover a very wide range of domains, including image recognition, speech recognition, language processing, and anomaly detection. Machine learning systems are becoming more and more prevalent in modern society, such as in search engines, video recommendation functions, security cameras, smartphones, and automated driving. Conventional rule-based pattern recognition by humans has limited recognition accuracy to process data existing in the natural world as it is. Machine learning methods for self-recognition of potential patterns from big data have achieved recognition accuracy that could not be surpassed by conventional methods, and in some cognitive tasks, have surpassed the recognition accuracy of human cognition.

We believe that there are two major types of applications of machine

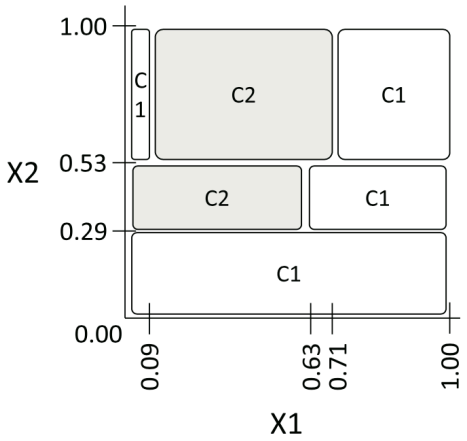


Figure 1.1: Example of a classification.

learning: server-side inference and front-side inference. Server-side inference, such as cloud computing and fog computing, often has small physical and power consumption constraints and requires overall system throughput. Front-side inference, such as edge computing and Embedded Systems, has very severe physical and power constraints. Embedded systems need to be small in size due to physical constraints, and the response time of the system is important because real-time processing is required. Self-sustaining systems, such as automated driving and robots, are embedded systems and require front-side reasoning with severe constraints. These autonomous systems are expected to be widely used in society in the future and will become increasingly important. This research investigates how to realize machine learning in Embedded Systems.

1.3.2 Random Forest

A decision tree is a popular method for various machine learning tasks. When a tree is grown very deep to learn highly irregular patterns, it overfits training sets. In that case, it has a low bias, while it has a very high variance. Fig. 1.2 shows an example of a decision tree that classifies a data set shown in Fig. 1.1. In Fig. 1.2, X_i denotes a feature variable for the dataset, and C_i denotes a label. Since the decision tree partitions the data set with a single feature variable, it often misclassifies out-lier labels. A random forest (RF) [5] is an ensemble learning method for classification and regression, and it consists of multiple decision trees. At learning, each decision tree is built by different (randomized) sub-sampling data from the same training set in order to reduce the variance. Fig. 1.3 shows an example of the RF, which consists

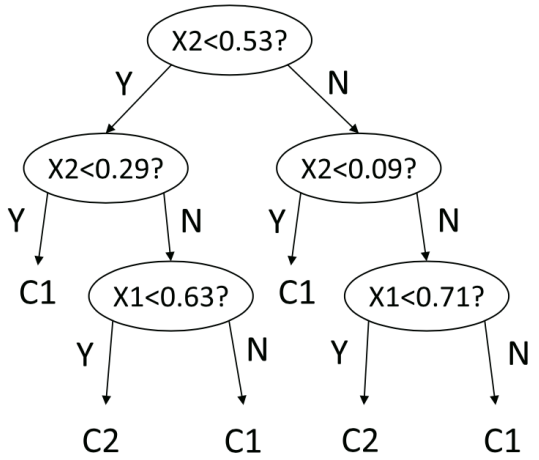


Figure 1.2: Example of a decision tree.

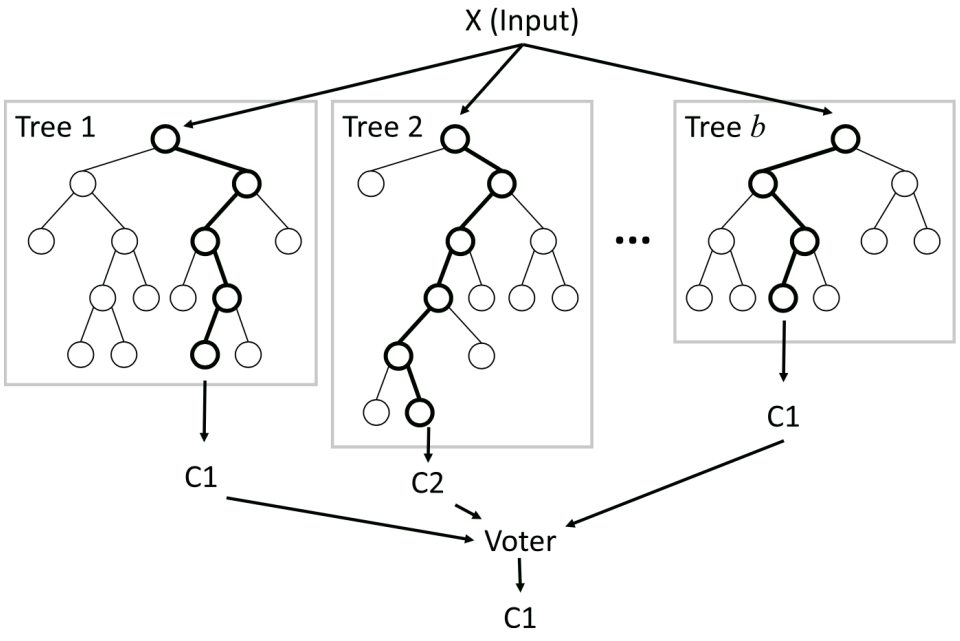


Figure 1.3: Example of a random forest.

of b decision trees and a voter. At first, decision trees branch corresponding to given feature variables. Then, they output the matched label. Next, the voter performs a majority decision of the labels from decision trees. Finally, it detects the most frequent label as a classification result. Since the RF uses training feature variables selected at random sampled, decision trees with low correlation are built. As a result, it improves accuracy and versatility.

The advantages of the RF are shown as follows [5]:

1. Classification accuracy is high, and it operates correctly even if the feature variables are from several hundred to thousands.
2. It is possible to estimate the importance of the feature variables for each label variable.
3. It effectively works with the dataset even if it lacks several feature variables.
4. The number of individual errors is maintained even in the unbalanced dataset.

On the other hand, the disadvantages are as follows:

1. Too deep decision trees fall into overfitting.
2. Classification accuracy is low with a small number of learning data.

In addition, classification accuracy is greatly affected by hyperparameters. The RF can be built with relatively appropriate hyperparameters using a grid search algorithm and encouraging parameters.

1.3.3 Deep Neural Network

Deep Neural networks are a type of machine learning that has received great attention due to their high accuracy. The formal neuron [6], the prototype of neural networks, was published in 1943. In 1986, backpropagation [7], an algorithm used to train neural networks, was proposed. In 1995, the convolutional neural network [8] was proposed, and image recognition was applied. However, at that time, neural networks did not attract as much attention as they do today due to the limitations of computing power.

Neural networks have been attracting attention again in recent years, mainly due to the GPU's improvement of computational power, which makes it possible to learn complex models. In particular, convolutional neural networks (CNNs) have received much attention for image recognition tasks since

AlexNet [9] won the 2012 competition. CNNs are known to increase recognition accuracy by increasing the number of layers. In VGG [10], we successfully trained a model with many layers by using the transition learning method. Res-Net [11] used residual learning to learn deep models and successfully trained a network with 152 layers. Since the model size is increasing faster than the computational power of computers, many efforts are also being made to improve the trade-off between computational cost and recognition accuracy. Mobile-Net [12] proposed a model that improves the trade-off between computational cost and recognition accuracy, focusing on embedded devices. SE-Net [13] introduced a lightweight computational squeeze-and-excitation structure to improve accuracy with little overhead. EfficientNet [14] quantitatively evaluates the computational complexity of network search and CNN models using reinforcement learning and clarifies the trade-off between recognition accuracy and computational complexity.

1.4 Objective

The objective of this research is to design an FPGA-based machine learning accelerator that satisfies the three constraints of inference speed, power consumption, and design cost. We show that optimizations that improve the trade-off between recognition accuracy and performance in machine learning can significantly improve the inference speed and power consumption constraints at the cost of a small amount of recognition accuracy. In addition, we propose a design flow using high-level synthesis, which makes it possible to easily implement the optimization method described above and to realize a high-performance accelerator implementation in a short period of development.

In Chapter 2, we will focus on random forests and propose to design a random forest using HLS. In this chapter, to further reduce the amount of hardware, we propose an optimization that uses k-means clustering to share the comparators of branching nodes on the decision tree. We have improved the trade-off between recognition accuracy and hardware usage for random forests by sharing the thresholds. We have reduced the design cost by proposing a series of design flows from model training to threshold sharing and hardware design for random forests. We implemented this random forest on a Xilinx FPGA and achieved a speedup of more than 8.4 times compared to the conventional method. In this chapter, we confirmed that the proposed optimization improves the trade-offs related to recognition accuracy and that the HLS-based design flow enables high-performance FPGA design in a short time.

In Chapter 3, we will focus on more practical tasks such as human pose estimation and CNNs, which have higher recognition accuracy. Since CNNs are computationally intensive, parallelization is not enough to speed up inference in real-time. In order to achieve faster inference, we focused on the sparsification of CNNs. Optimization by sparsification improves the trade-off between computational complexity and recognition accuracy of CNNs. We then propose an indirect memory access architecture for efficient convolutional operations of sparse CNNs. With this work, we have achieved about 3.5 times faster inference speed and about 13 times better power efficiency compared to the existing GPU method. We have greatly improved the inference speed and power consumption issues of CNNs, which consumed 55W.

In Chapter 4, we discuss how to realize CNNs on FPGA devices with smaller power consumption. There is a demand to achieve machine learning with smaller power consumption. CNNs have large weights and internal data called feature maps, which pose a challenge for FPGAs in terms of memory capacity. The problem of large weights has been solved by sparsification introduced in Chapter 3, but the memory capacity of feature maps is an issue. In order to solve this problem, we use Split-CNN, which splits the feature map and processes it in time. We have clarified the trade-off between recognition accuracy and hardware usage by splitting the Split-CNN. We designed a memory buffering method and scheduling for Split-CNN and implemented it on a PYNQ-Z1 FPGA board, a low-end FPGA with a power consumption of about 3W. With this achievement, we have succeeded in reducing the power consumption to less than one-third while achieving 3.1 times faster speed compared to GPU. We have improved the power consumption problem by realizing CNN on a low-end FPGA.

In Chapter 5, we proposed a CNN accelerator on a low-end FPGA with DSP utilization exceeding 90% and operator utilization efficiency exceeding 90%. The computational power of a computer is determined by the degree of parallelism, the efficiency of operator usage, and the operating frequency. Now that we have implemented a design with sufficiently high parallelism and efficiency, we need to increase the operating frequency to realize faster inference on FPGAs. Designing large circuits with HLS has the problem of decreasing the operating frequency. Designing with HLS also has a problem of scaling to large circuits. We will consider using multiple small circuit modules designed using high-level synthesis to achieve a high operating frequency. We propose an architecture using a ring bus and computation scheduling for CNNs to hide the communication overhead caused by partitioning the circuits. In order to easily map complex models by partitioned circuits, we have developed an instruction set architecture dedicated to CNNs and developed a deep learning compiler for the proposed architecture to solve the design

cost problem. When implemented on a Xilinx FPGA board ZCU102, we were able to achieve an operating frequency of 500MHz despite the relatively large design occupying more than 140k LUTs and 800 DSPs.

Through this research, we have achieved an FPGA-based accelerator that simultaneously improves the three constraints of inference speed, power consumption, and design cost by using an HLS-based design method and optimization with a slight sacrifice of recognition accuracy.

Chapter 2

An FPGA Realization of a Random Forest with k-Means Clustering using HLS

2.1 Introduction

2.1.1 Acceleration of the Random Forest

A decision tree is a popular method for various machine learning tasks. When a tree is grown very deep to learn highly irregular patterns, it over-fits training sets. In that case, it has a low bias, while it has a very high variance. Since the decision tree partitions the data set with a single feature variable, it often misclassifies outlier labels. A random forest (RF) [5] is an ensemble learning method for classification and regression, and it consists of multiple decision trees. As for training, each decision tree is built by different (randomized) samples from the same training set. Decision trees with low correlation are built since the RF uses training feature variables selected at random sampled. As a result, it improves accuracy and versatility compared with a single decision tree-based classification. The RF is widely used for classification. For example, they are a key point matching [15], a segmentation [16], a pedestrian detection [17, 18], a human pose estimation [19], a face direction estimation [20], and an IP address search for the Internet [21]. These applications are demanded to be recognized in real-time. However, since the classification speed in the CPU is too slow, hardware acceleration is necessary. Also, low power consumption is desired since it is often used in embedded systems. However, a single instruction multiple data (SIMD) architecture, typified by GPU, is not suitable for the RF with three reasons as follows:

1. Higher precision:

   Each node in the decision tree can be evaluated by if-then-else statements to evaluate decision trees in the RF. A conditional expression in the if-then-else statement compares an input with a constant value, which is represented by a floating-point representation. Although the GPU supports a double-precision floating-point, such high precision is not required for the RF classification. Therefore, a high-precision arithmetic circuit is inefficient in hardware and power consumption.

2. Uniformly processing (CUDA) cores:

   The GPU runs the SIMD operations, having a large uniformly processing core. These cores are specialized in parallel data computation. However, the RF consists of decision trees of a different size, which causes an unbalanced computation. Since a warp divergent frequently occurs, computation time would be bound by the decision tree, the longest path.

3. Higher cost for the all-to-all communication:

   The GPU can be performed at a relatively high-speed communication between near processing cores with the same local memory. In contrast, its communication penalty is large for all-to-all communication. All-to-all communication would always occur since an RF requires the whole majority detection after evaluating all the decision trees.

2.1.2 FPGA Realizations

Since the FPGA can configure a dedicated all-to-all communication circuit, it is possible to configure FPGA heterogeneous cores with different decision tree sizes and appropriate variable bit length circuits. Thus, the FPGAs are suitable for accelerating the RF. Becker et al. used decision trees to accelerate object tracking. They focused on heavily parallelizing the classification and converted the input data into a dedicated representation in the FPGA [22]. Essen et al. showed a pipelined architecture and a single instruction multiple threads (SIMT) algorithm for the RF on FPGA. Also, they compared the FPGA-based implementation with the multicore CPU and the GPU [23]. Oberg et al. implemented the RF on the FPGA with the Kinect depth-image sensor for the Forest Fire pixel classification algorithm [24]. However, the conventional realizations are designed by the RTL description. Compared with the software-based design, it takes an enormous amount of development time [25]. As for the RF design, since its structure is

completely different for each dataset, it is not practical to tune the architecture by the RTL description. In recent years, many high-level synthesis tools have been proposed to reduce the development time. A typical high-level synthesis supports the C/C++ codes as input design. Especially, the Intel SDK for OpenCL [2] and the Xilinx SDSoC [26] are system-level design tool that includes system level profiling, automated software acceleration in programmable logic, automated system connectivity generation, and libraries to speed programming. Therefore, it can accelerate the target application with a short development time. In this chapter, we accelerate the RF on the FPGA. An acceleration method of the RF for training using the FPGA has already been proposed [27]. In this chapter, we assume that the learning is done offline. We compare the software-based design such as a CPU and a GPU for the classification time and power consumption efficiency.

2.2 High Performance and Short Time Design

2.2.1 Comparison of Computation Model

In recent years, Altera Corp. has promoted the Altera SDK for OpenCL for the FPGA development environment, and Xilinx Inc. also has released the SDSoC. These tools include system-level profiling, automated software acceleration in programmable logic, automated system connectivity generation, and libraries to speed programming. The GPU programming model is applied directly provides the familiar C/C++ application development experience. Since the target computation models are different, it is hard to accelerate the application even if we used such a system-level tool.

Here, we explain the programming model for the FPGA fitted system and the GPU fitted one. Fig. 2.1 shows an architecture model for the GPU, while Fig. 2.2 shows that for the FPGA. For the GPUs, it runs the threads in the workgroup to the data-parallel model by using a large number of CUDA cores and wideband data transfer memory such as DDR5 off-chip memories. On the other hand, for the FPGA, since data communication bandwidth, for off-chip memory is narrow, it tends to configure the pipeline model in the workgroup. When the pipeline stall is free, communications with the off-chip memory are only input and output of the pipeline. Therefore, the FPGA can realize a high-throughput operation even if it has a narrow band to the off-chip memories. Fortunately, since the FPGA has more on-chip memories than the GPU, data transfer between the pipeline stages on the FPGA is possible. The modern system design tools for the FPGA provide a channel

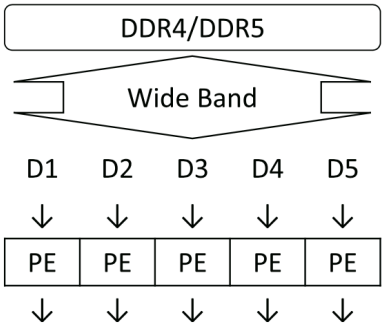


Figure 2.1: Computation model for the GPU.

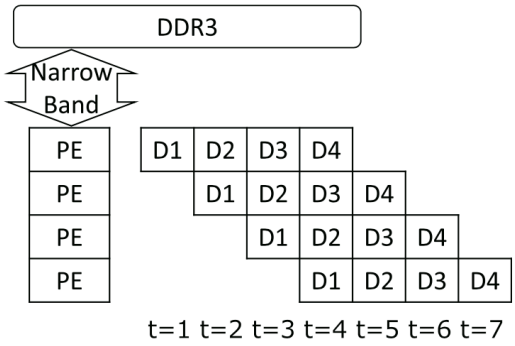


Figure 2.2: Computation model for the FPGA.

to make an on-chip communication between pipeline stages. Furthermore, another advantage of the FPGA is that it is possible to realize a customized pipeline stage (in other words, it can realize a heterogeneous core) in parallel. When the latency of each parallel operation is different, its computation time for the GPU would be bound by the longest one. On the other hand, the hardware resources in the FPGA are appropriately distributed, it is possible to realize a heterogeneous parallel architecture with uniform latency.

2.2.2 Short Time Design Using HLS

Another feature of the system design tool is a short design compared with the conventional HDL design. The modern system-level design, such as an Intel SDK for OpenCL and a Xilinx SDSoC, supplies the board support package (BSP) for recommended FPGA boards. The BSP prepares the IP cores and external memories to bridge the host program and the kernel program on the host processor. For the conventional FPGA design, since the programmer designed them in the RTL description, it could not respond to frequent changes in long-term design. Using a system-level design can generate the configuration data that automatically connects with the user program. Therefore, since it is possible to reduce design time remarkably, the programmer can concentrate on tuning architectures. The remaining problem for the RF design on the FPGA is the C/C++ code refactoring, which consumes design time. In the chapter, we propose the *krange* tool flow, which automatically generates optimized C/C++ code to reduce the design time.

2.3 Conventional Techniques

2.3.1 Fixed-Point Representation

As shown in Fig. 2.2, the bandwidth between the off-chip memory and the kernel on the FPGA becomes a bottleneck. If-then-else statements can express each node in a decision tree. For many random forest software libraries, a conditional expression in the if statement compares a feature variable with a constant value represented by a 32-bit floating-point representation. We use an n-bit signed fixed-point representation instead of a 32-bit floating-point. Fig. 2.3 shows the pseudo-codes for the decision tree using a floating-point and 14-bit signed fixed-point representation. Note that, in the right side pseudo code, the most signification bit (MSB) represents a sign, while the latter bits represent a number. Although the standard C/C++ supports only 8, 16, 32, and 64-bit integers, modern high-level synthesis tools sup-

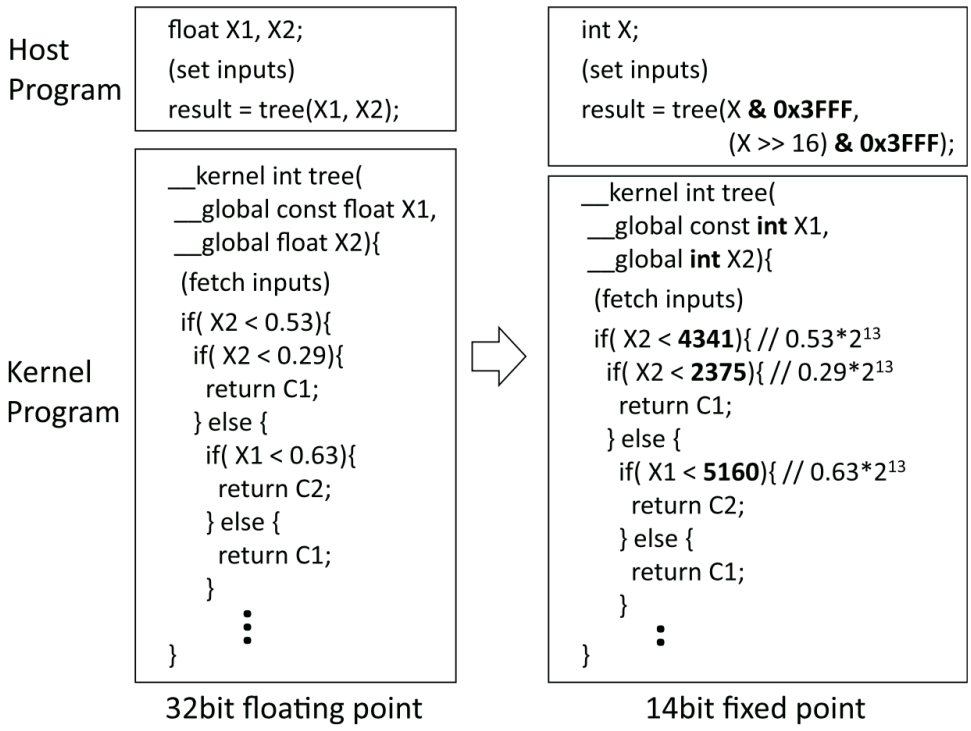


Figure 2.3: Example of a fixed-point representation.

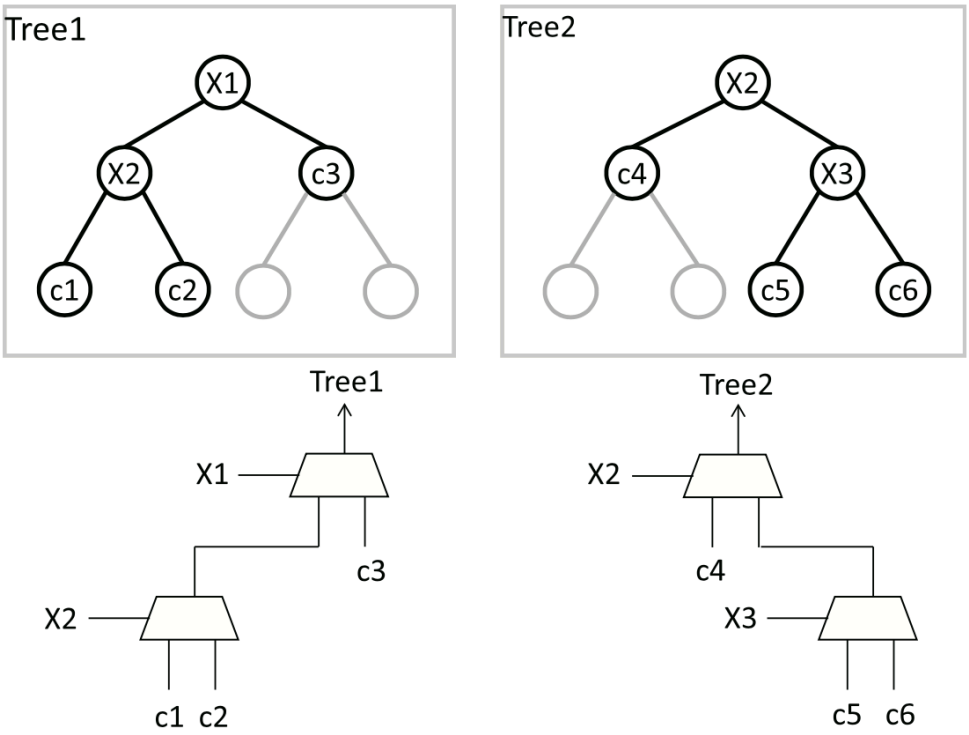


Figure 2.4: Example of a multiplexer tree realization for a decision tree.

port variable bit integers using an appropriate mask. Depending on the bit width, a fixed-point representation lacks accuracy compared with a floating-point one. Thus, a fixed-point representation may cause misclassification. However, since it compresses the off-chip memory bandwidth to n, it can accelerate the classification. Furthermore, since multiplexer trees for a fixed-point are simplified, they are faster and smaller than floating-point-based multiplexers.

2.3.2 Pipeline Stages by Loop Unrolling

We connect decision trees in series to form a deep pipeline with on-chip memory. It increases the system throughput. Also, we realize the voter by the pipeline circuit. Fig. 2.4 shows an example of decision trees and their hardware realization by a multiplexer tree. When the decision trees are written by a for statement, as shown in Fig. 2.5, the high-level synthesis tool sequentially traces decision trees by a shared multiplexer circuit. Conversely, we can increase the throughput by using an #pragma unroll, expanding a sequential circuit to a pipelined one. Fig. 2.6 shows a pipeline circuit with an

```
__kernel int RF(
 __global float X1, X2, X3){
 (fetch inputs)

 for( int i = 0; i < 2; i++){
  if( i == 0) class = tree1(X1,X2,X3);
  else class = tree2(X1,X2,X3);

  voting[class]++; // voter
 }
        :
}
```

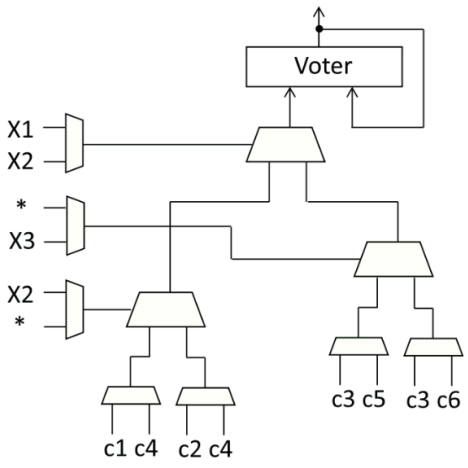


Figure 2.5: Sequential realization.

```
__kernel int RF(
 __global float X1,X2,X3){
 (fetch inputs)
 #pragma unroll 2
 for( int i = 0; i < 2; i++){
  if( i == 0) class = tree1(X1,X2,X3);
  else class = tree2(X1,X2,X3);

  voting[class]++; // voter
 }
      :
}
```

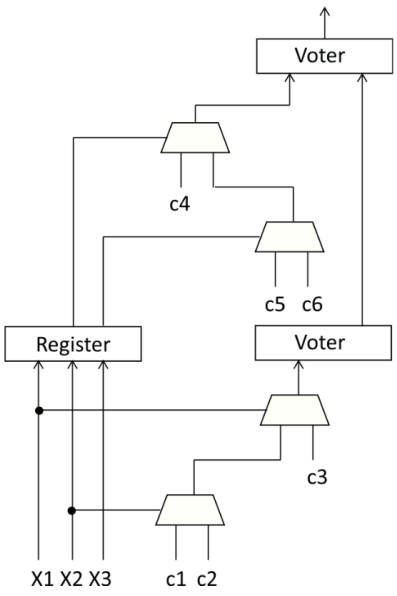


Figure 2.6: Pipeline realization with an unroll pragma.

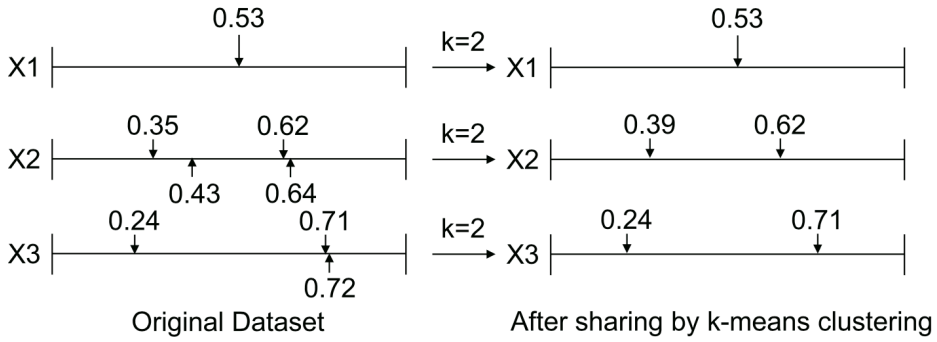


Figure 2.7: Example of k-means clustering.

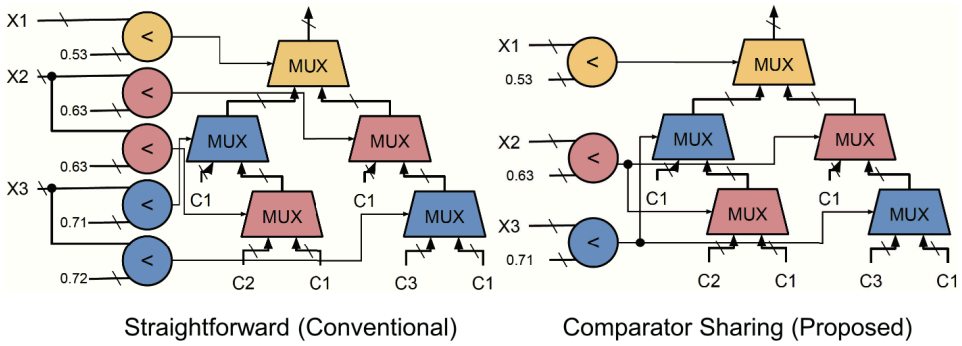


Figure 2.8: Example of comparator sharing by k-means clustering.

#pragma unroll. Registers and voters are inserted between the multiplexer trees in the pipeline circuit. In that case, the number of registers and the memories tends to increase, and the on-chip memories realize them. Since communication with the off-chip memory does not occur, it can accelerate throughput.

2.4 Comparator Sharing by *k*-Means Clustering

2.4.1 Hardware Reduction

A typical approach to reducing the amount of hardware is resource sharing. In the RF, a comparison of the input variable with the threshold is required in each decision tree node. They can be shared by summarizing comparisons with similar thresholds for certain input variables. Each decision tree node

compares the input variable with the constant threshold. In other words, it divides the feature space by the constant threshold. Since to summarize similar comparisons is the same as roughening the division of feature space, it is expected that the classification accuracy may be worse. Thus, appropriate sharing is necessary to find small hardware with classification accuracy. We share similar comparisons using the k-means method, a kind of clustering algorithm. Fig. 2.7 shows an example of k-means clustering. In other words, it reduces the number of thresholds for feature space partitioning. By applying k-means clustering, comparators can be merged into a single one. Fig. 2.8 shows an example of comparator sharing by k-means clustering.

2.4.2 *k*-Means Clustering

A k-means clustering [28] is one of the simplest unsupervised learning algorithms. The procedure classifies a simple and easy to classify a given data set by using a certain number of clusters (assume k clusters) following simple and easy ways. The main idea is to define a center value for each cluster. These k centroid values should be placed cleverly since different location causes the different result. Thus, the better choice is to place them as much as possible far away from each other. The next step is to take each point belonging to a given data set and associate it with the nearest centroid. When no point is pending, the first step is completed, and an early grouping is done. Re-calculation for k new centroids is performed as barycenters of the clusters resulting from the previous step. After these k new centroids are obtained, a new binding has to be done between the same data set points and the nearest new centroid. k centroids are changed their location step by step until no more changes are done. In other words, centroids do not move anymore. Finally, this algorithm aims at minimizing an objective function. Let n be the number of data points in the cluster. In this case, a squared error function F_{err} is defined as follows:

$$F_{err} = \sum_{j=1}^{k} \sum_{i=1}^{n} \|x_i^{(j)} - c_j\|^2$$

where $\|x_i^{(j)} - c_j\|^2$ is a distance measure between a data point $x_i^{(j)}$ and the cluster center c_j. It indicates the distance of the n data points from their respective cluster centers.

The algorithm consists of the following steps:

Algorithm 2.1:

1. Place k points into the space represented by the objects that are being

clustered. These points represent initial group centroids.

2. Assign each object to the group that has the close st centroid.

3. When all objects have been assigned, recalculate the positions of the k centroids.

4. Repeat Steps 2 and 3 until the centroids are no longer changed. It produces a separation of the objects into groups from which the metric to be minimized can be calculated.

An advantage of the k-means method is that its algorithm is simple and operates at high-speed. The result of clustering depends on the random initial value for centroids. [29] proposed devising the initial allocation for centroids. Since it is necessary to give the number of clusters k, it uses other indicators to select the optimum clusters. An estimation method for the optimum number of clusters using the k average method [30] has been proposed.

A similar comparison operation of the random forest is shared using k-means clustering. The algorithm for the comparator sharing RF is shown as follows:

Algorithm 2.2:

1. Thresholds of nodes are collected for each feature variable at the training.

2. The thresholds recorded for each feature variable are clustered by the k-means method and classified into k clusters, respectively.

3. Each centroid is used to compare the corresponding input feature variable at the inference.

In the RF, each node consists of a comparator and a multiplexer. Since the centroid is used as the threshold for branching of the multiplexer, almost initial comparators can be shared. Therefore, it is possible to drastically reduce the comparator in the random forest.

2.4.3 Tool Flow

Fig. 2.9 shows the proposed tool flow, which uses *krange* (k-means based random forest generator). First, we use the scikit-learn software [31] to learn the RF from a given dataset. Note that we find the optimum hyperparameter set by a grid-search algorithm. Then, we shared the threshold by k-means

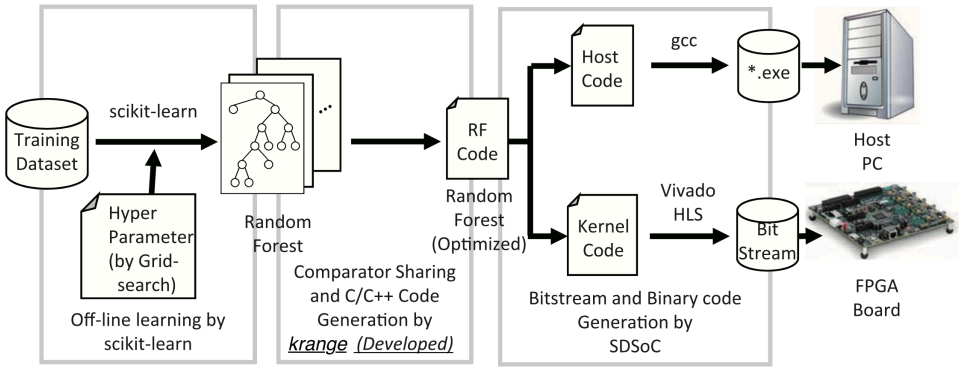


Figure 2.9: Proposed tool flow.

Table 2.1: Dataset used in the experiment.

Dataset			
Name	#Rules	#Feat.	#Class
Arrhythmia	452	279	16
Dermatology	366	33	6
Ionosphere	351	34	2
Iris	150	4	3

clustering with hyperparameters, the number of clusters. Next, we generate the host code and the kernel code for the high-level synthesis tool for the FPGA. The generated codes are converted into the bitstream file using the logic synthesis tool. Since the proposed tool flow automatically generates the bitstream from a given dataset, we can concentrate the parameter tuning to accelerate the RF.

2.5 Experimental Results

2.5.1 Implementation Environment

We implement the UC Irvine machine learning repository [32] to a Xilinx Inc. ZC702 evaluation board with a Xilinx Corp. Zynq7020 (53,200 LUTs, 140 36KbBRAMs, 220 DSP blocks). To generate an executable code, we used the proposed tool flow shown in Fig. 2.9. For the host PC, we used Intel ' s Xeon (R) E5607 Processor (2.26GHz, four cores) with 32GB DDR3 off-chip memory and Ubuntu 14.04 LTS (64-bit version) operating system. Table 2.1 shows the used dataset, and Table 2.2 shows their parameters. Note that, to

Table 2.2: Hyperparameters used in the experiment.

Hyper Parameters					
Name	#Trees	Depth	#Feat.	k	PS
Arrhythmia	35	20	20	4	20
Dermatology	30	5	7	1	5
Ionosphere	25	15	10	4	15
Iris	50	20	2	4	20

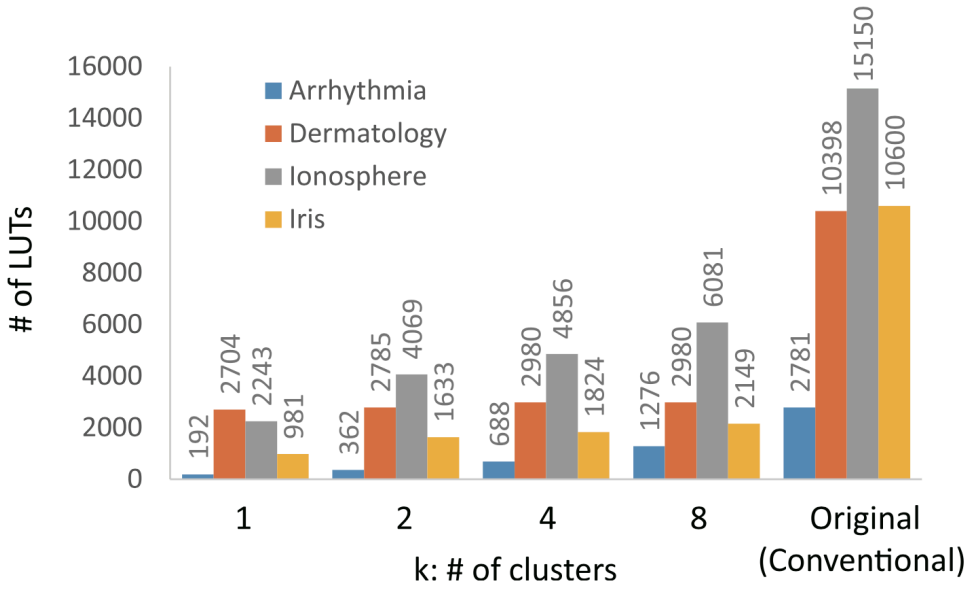


Figure 2.10: Comparison of the number of LUTs.

find the optimum parameter set, we used a grid search algorithm available in scikit-learn. We showed the dynamic power consumption. We measured both the static power consumption and the total power consumption to compute random test vectors. Then, we obtained the dynamic one.

From the previous work [33], since the 14-bit fixed-point precision caused no classification error degradation, we used such custom precision to the RF implementation. We inserted pragma unroll at the top of each loop in the implementation. Thus the number of pipeline stages equals the depth for each decision tree. In Table 2.1, we showed the pipeline stages as (PS). Also, we showed the number of clusters k used in the experiment in Table 2.2.

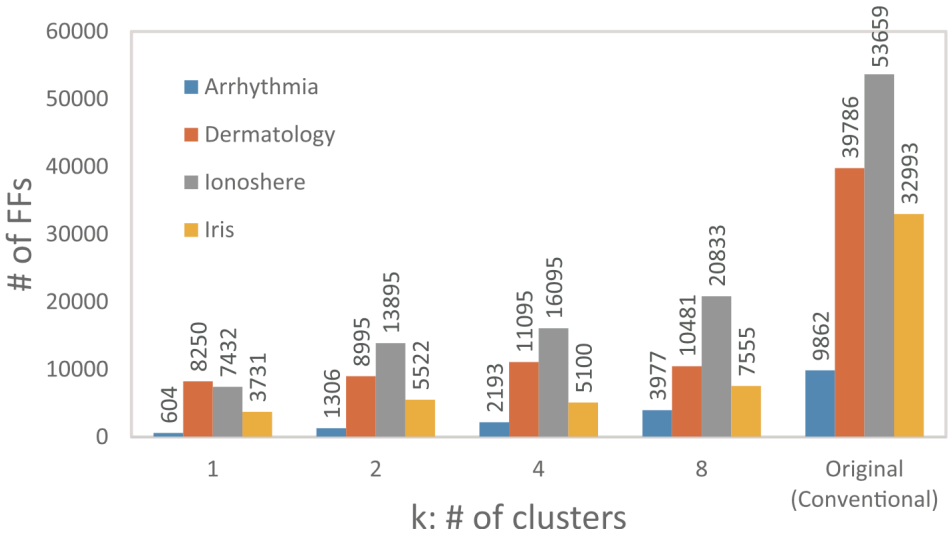


Figure 2.11: Comparison of the number of flip-flops (FFs).

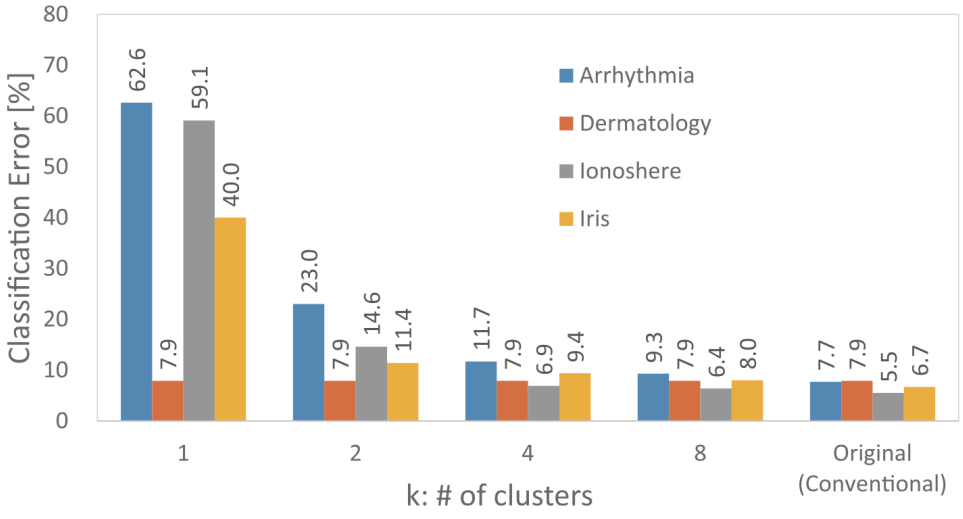


Figure 2.12: Comparison of classification error (%).

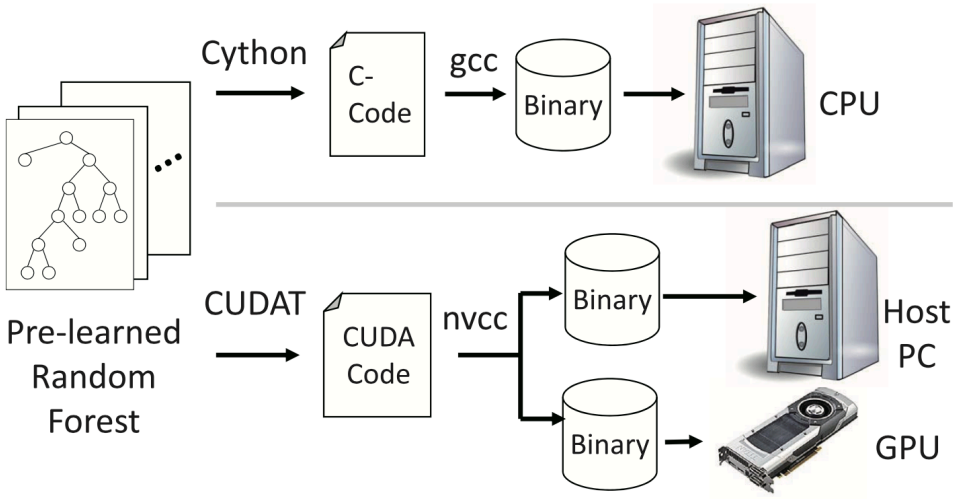


Figure 2.13: Execution flow for the CPU and the GPU.

2.5.2 Compared with Conventional Methods

We set the number of clusters k to 1, 2, 4, and 8, respectively. Then we measured the number of comparators (thresholds) and the classification accuracy. We compared the proposed comparator sharing with the conventional realization [33], which did not share the comparators. Fig. 2.10 compares the number of LUTs, which is a bottleneck of the RF implementation and Fig. 2.11 compares the number of flip-flops (FFs). Fig. 2.12 compares the classification error rate. As shown in Fig. 2.10, Fig. 2.11 and Fig. 2.12, there is a trade-off between the hardware consumption and the classification accuracy. It is depended on the variation of a dataset. In Fig. 2.12, even if we decreased k, the classification error for Dermatology did not increase. Since its feature values exist around similar values, the clustered value took almost the same value. For another dataset, when $k = 2$, their error rates are considerable, and when $k = 4$, those rates are slightly increased. Thus, we think that $k = 4$ is practical value for all cases. The above discussions showed that we should carefully consider the error rate when designing the RF on the FPGA using a clustering method since it depends on its dataset. The experiment reduces the hardware resource usage by 41% at maximum with a 1% error reduction of the baseline accuracy, while 64% reduces it at one with 5% or less. Thus, the k-means clustering efficiently reduced the amount of hardware.

Table 2.3: Comparison with the CPU and the GPU.

	GPU@86W 876 MHz Geforce Titan		CPU@13W 2.26 GHz Xeon (R) E5607		FPGA@14W 100 MHz Zynq7020	
Name	LPS	LPS/W	LPS	LPS/W	LPS	LPS/W
Arrhythmia	33.6	0.52	211.6	16.27	65.7	4.69
Dermatology	71.8	0.84	488.4	37.57	3270.0	233.50
Ionosphere	82.1	0.95	595.9	45.84	3165.0	226.10
Iris	44.7	0.52	436.7	33.59	8087.0	577.60
Ratio	0.016	0.003	0.119	0.128	1.000	1.000

2.5.3 Compared with Other Platforms

We compared the FPGA with the CPU and the GPU for the lookups per second (LPS) and the power consumption efficiency (LPS/W). Fig. 2.13 shows the execution flow used in the experiment. As for the CPU platform, we used Intel ' s Xeon (R) E5607 Processor (2.26GHz, four cores) with 32GB DDR3 off-chip memory and Ubuntu 14.04 LTS (64-bit version) operating system. To generate the executable code, we first generated the RF by the scikit-learn with the same parameters shown in Table 2.1. Then, we converted the RF to C-codes by Cython [34] and compiled it into executable code by the GCC compiler. As for the GPU platform, we used the NVidia Geforce Titan (876 MHz, 2,496 CUDA cores, and 6GB DDR5 off-chip memory) with the same processor and the main memory running on the Ubuntu 14.04 LTS. To generate the executable code, first, we generated the RF by the scikit-learn, then used the CUDA Tree (CUDAT) [35] to generate the executable code. To measure the LPS, we used 1,000 random test vectors, while to measure the power consumption excluding the idle power, we inserted the power measurement between the host PC and the power source. As for the FPGA realization, we set appropriate numbers of bits (14-bit fixed-point precision) and unrolls from the above experiments. On the other hand, for both the CPU and the GPU, 32-bit floating-point precision. Since the FPGA only can realize a custom bit-length precision to realize a high-performance circuit, it is an advantage to use the FPGA.

Compared with the previous implementation in [33], the present result is worse since the previous implementation used the high-end FPGAs to use at the data center. In contrast, this implementation used the low-end FPGAs to the embedded system for a low-end system.

Table 2.3 compared the CPU and the GPU realizations. We used the best k, which reduced maximum hardware resource, and its classification

error reduction was around 1% of the baseline accuracy. As shown in Table 2.3, the FPGA realization is 8.4 times faster and 7.8 times better power efficiency compared with the CPU. Compared with the GPU, it is 62.8 times faster and 385.9 times better power efficiency. Since the RF requires heterogeneous if-then-else statements, the conventional homogeneous architectures are unsuitable for such applications. Only the FPGA can configure custom pipelined architecture for branch operations in the RF. Thus, it achieved higher performance and lower power consumption.

2.6 Conclusion

This section showed the acceleration method for the RF on the FPGA. We proposed a fully pipelined architecture to accelerate the RF, including an all-to-all communication circuit. It increased the memory bandwidth using on-chip memories on the FPGA. To further improve the RF for the FPGA realization, we used k-means clustering to share the comparator of the decision tree on the RF. We also developed the *krange* tool flow, which generates the bitstream with only a few hyperparameters. We implemented the UC Irvine machine learning repository on the Xilinx Inc. ZC702 evaluation board. Compared with the CPU and the GPU realizations, as for the LPS, the FPGA realization was 8.4 times faster than the CPU one, and it was 62.8 times faster than the GPU one. As for the LPS per power consumption, the FPGA realization was 7.8 times better than the GPU one, and it was 385.9 times better than the GPU one.

Chapter 3

An FPGA Realization of OpenPose with a Sparse Weight Convolutional Neural Network (CNN)

3.1 Introduction

3.1.1 Pose Estimation

Human 2D pose estimation, the problem of localizing anatomical key points or parts, has focused mainly on finding body parts of individuals. Inferring the pose of multiple people in images, especially socially engaged individuals, presents a unique set of challenges. First, each image may contain an unknown number of humans that can occur at any position or scale. Second, interactions between people induce involved spatial interference due to contact, occlusion, and limb articulations, making an association of parts difficult. Third, runtime complexity tends to grow with the number of people in the image, making real-time performance a challenge.

OpenPose [36, 37, 38] uses a high-performance GPU. Thus, power consumption becomes a critical issue to realize on embedded devices. Also, its computation time is too slow than the current video standard frame speed. This section implements an efficient multiperson pose estimation method with state-of-the-art FPGA accuracy. We use the OpenPose pose estimation algorithm, the first bottom-up representation of association scores via Part Affinity Fields (PAFs), a set of 2D vector fields that encode the location and orientation of limbs over the image domain. It takes the entire image as the

Figure 3.1: OpenPose overview.

input for a two-branch CNN to jointly predict confidence maps for body part detection. Fig. 3.1 shows an overview of the OpenPose process. It detects body parts and estimates parts associations at a time. Then, it connects associate body parts from candidates. It finally assembles them into full-body poses for all people.

3.1.2 Sparseness for a Large CNN

As for the recurrent neural network [39] and the CNN [40], Nurvitadhi et al. compared the FPGA implementation with the CPU, the GPU, and the ASIC, and it showed that the FPGA could deliver orders of magnitude improvements in performance and performance/watts over well-optimized software on CPU and GPU. Although FPGA is less efficient than ASIC, the FPGA-ASIC gap may be reduced for designs that heavily utilize hard blocks [41]. Hence, FPGA offers an attractive solution, which delivers superior efficiency improvements over software without having to lock into a fixed ASIC solution.

We show that the computationally intensive part of the OpenPose algorithm is a convolution, and its memory size is too large to store the modern FPGA. We reduced the memory size for weights by introducing sparseness techniques. In the weight sparseness CNN, an internal weight can take $-w$, $+w$, and zero with a given threshold for an inference, where a skip computation can realize zero weight. Also, we propose a multi-stage pipelined convolution by inserting a buffer memory to increase system throughput.

3.2 Sparse Weight CNN

3.2.1 CNN for OpenPose

The CNN for OpenPose generates confidence maps for part detection and a parts association. The former role used a fine-tuned VGG19 CNN to extract a feature from an incoming image. In the latter part, it uses the two-branch multi-stage CNN. Each stage in the first branch anticipates confidence maps, and each step in the second branch predicts PAFs. After each stage, the predictions from the two branches and the image features are concatenated for the next stage.

Even if we increase the number of stages, accuracy is slightly improved. As for the accuracy and hardware trade-off, we set $t = 2$. Since the baseline OpenPose CNN requires 14,808 18Kb BRAMs, it is impractical for an on-chip memory realization on a modern FPGA. Thus, compression of the weight memory must be considered. For the execution on the high-end Titan X GPU (Pascal architecture), its frames per second (FPS) is around 7, which is slower than the standard real-time video FPS. Its power dissipation is more than 200 W. Therefore, a real-time computation with a low-power consumption accelerator is desired. One challenge is to realize the OpenPose using a sparse weight CNN on an FPGA. Since it can reduce the amount of computation and the memory size, we can store all parameters on an FPGA. It allows us to eliminate energy-costly DRAM access with a high-bandwidth [42]. From profile analysis, the CNN part is computation intensive. Thus, we realize an FPGA accelerator for the OpenPose. Other parts are executed on the host PC.

3.2.2 Convolutional Operation

A 2D convolutional operation applies the multiply accumulation (MAC) operation to the feature map's $K \times K$ size kernel. It dramatically reduces the number of parameters involved, allows local features, and avoids over-fitting. The output $Z^{(i)}_{l,r,c}$ of the i convolutional layer, which takes input N_i images (feature maps) of dimension $K \times K$ at location (r, c), is calculated as follows:

$$Z^{(i)}_{l,r,c} = f_{act}\left(\sum_{s=0}^{N_i}\sum_{j=0}^{K}\sum_{l=0}^{K} W_{i,j,l,s} X_{i-1,s,r+j,c+k}\right)$$

where $K \times K$ is the dimensions of the kernel for the convolution operation.

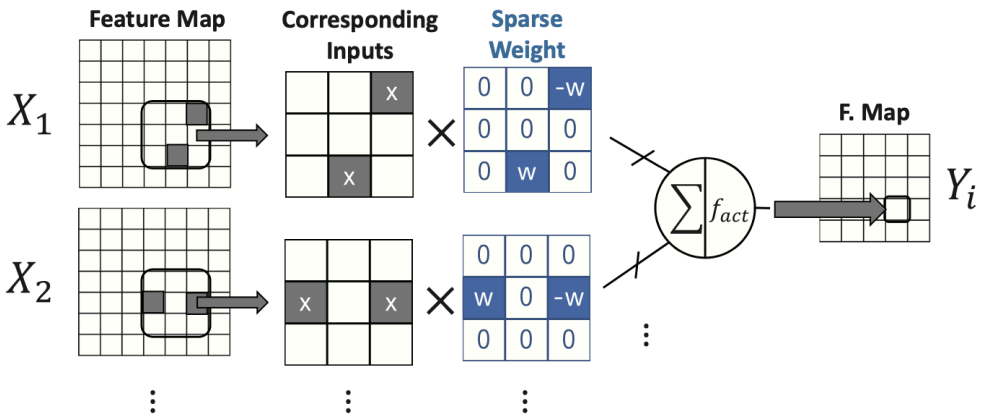


Figure 3.2: Example of a sparse convolution.

3.2.3 Weight Sparseness

We introduce the sparse weight CNN, which is suitable for hardware implementation. We consider that the sparse weight consists of $-w$, $+w$, and zero. The sparse weight CNN has hidden weights $w^{(hid)}$ during the training on the GPU. We defined the sparse weight $w^{(t)}$ from the hidden weight as follows:

$$w^{(t)} = \begin{cases} 0 & |w^{(hid)}| <= \rho \\ w^{(hid)} & |w^{(hid)}| > \rho \end{cases}$$

where ρ denotes the threshold to distinguish a zero-weight and a non-zero one, we hardly set ρ.

3.3 Sparse Weight Architecture

Fig. 3.2 shows an example of a sparse convolution operation. In the baseline CNN, its weight always takes a value of $-w$ or $+w$. Thus, there is no state that neurons disconnect with each other. In the sparse weight CNN, we define the state of the weight zero, which is possible to represent disconnections. The matrix representation of the sparse weight CNN is a sparse one, so we can apply the sparse matrix operations to reduce the amount of computation.

As shown in the previous chapter, when the sparse weight becomes zero, the output becomes zero. Therefore, we can skip the MAC operation for the zero-weight [43]. The target platform is the FPGA which can configure multiple memory access architecture with a custom data structure. We realize the sparse weight convolutional operation by zero-weight skip computation. Fortunately, since the pre-trained CNN has zero-weight, we store the

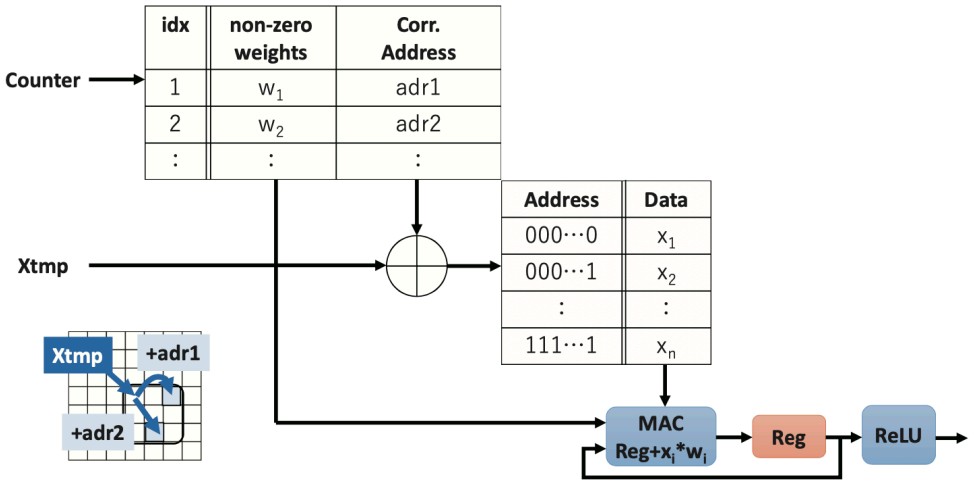


Figure 3.3: Indirect memory access for a sparse convolution.

address corresponding to the non-zero-weight. Thus, we realize the sparse weight convolutional operation by the conventional CNN operation with zero-weight skip one. The proposed 2D convolutional operation requires L words, where L denotes the number of non-zero-weights. Since the CNN with many zero-weights requires fewer memory accesses, it is faster than the straightforward 2D computation.

The zero-skip computation can be done by using indirect memory access. Fig. 3.3 shows indirect memory access for a sparse convolution. First, it reads a non-zero-weight and a corresponding address at a time. Then, it computes an address for a corresponding input. Next, it reads a corresponding one then performs the MAC operation. Repeating the above operations for all non-zero-weights in the kernel applies the activation function (In Fig. 3.3, it is a ReLU).

We propose a kernel parallelization for the sparse weight convolutional operation to increase throughput further. Fig. 3.4 shows a kernel parallelization for the sparse weight 2D convolutional operation. Since the convolutional operation uses the same kernel, the same weight is applied to the MAC operation in the same row in the feature map. We can compute the MAC operation in parallel since the internal outputs Y are independent. We load a line in the sparse weight CNN when the non-zero-weight overlaps.

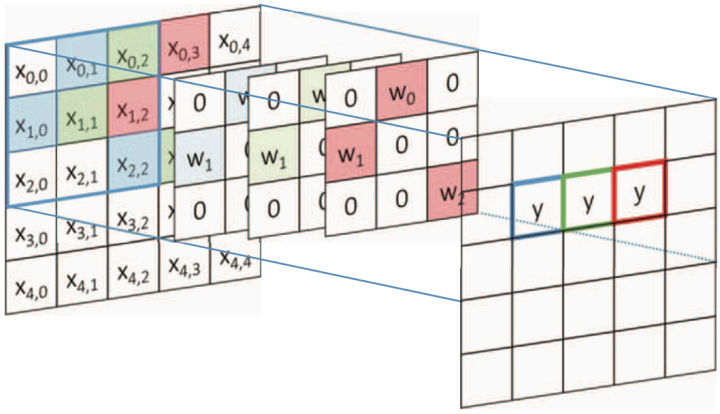


Figure 3.4: Kernel parallelization for the sparse weight 2D convolutional operation.

3.4 FPGA Implementation

3.4.1 High-Throughput Multi-Stage CNN

We apply the multi-stage pipeline architecture to keep up with the real-time pose estimation. We attached a feature map memory that supports stride access for multiple feature map memory access. We realize a large feature map memory by composting on-chip memories on the FPGA. Thus, the feature map memory accepts high-bandwidth memory access. The feature map memory is inserted between pipeline stages to realize a buffer memory.

3.4.2 Overall Architecture

We schedule the pipeline computation from the MAC operation analysis. As a result, we split the OpenPose CNN convolutional operation into six stages. Fig. 3.5 shows an overall architecture with six sparse convolution units with a feature map memory as a pipeline buffer. All the memories are stored in on-chip memories on an FPGA. Fortunately, the recent Xilinx Virtex Ultrascale+ device equips the UltraRAM, larger than the block RAM (BRAM). We use UltraRAMs to realize a feature map memory for the first layer buffer on the VGG19 part. It is a kind of stage pipeline architecture with various units. It is suitable for the FPGA realization, not for the GPU.

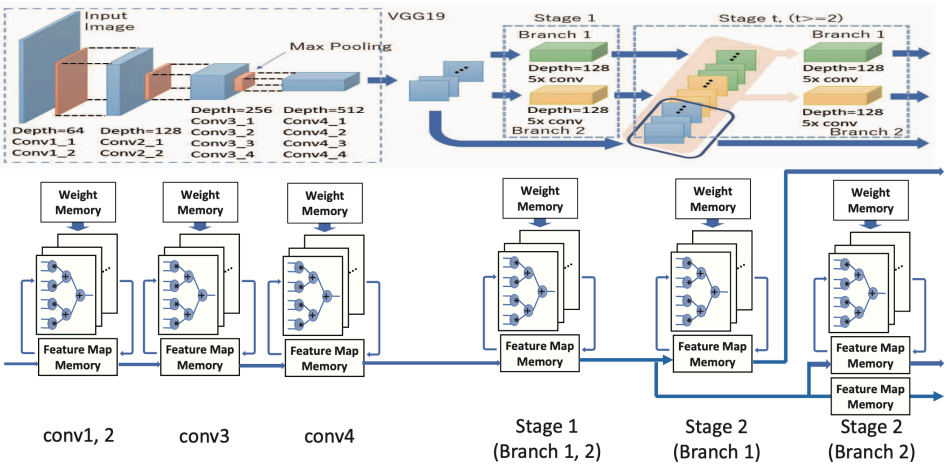


Figure 3.5: Overall architecture.

3.5 Experimental Results

3.5.1 Training Results

We implemented the weight sparse OpenPose CNN training script using Python 3.5 and a Chainer deep learning framework version 2.1.0. We considered that the input image size was 368 × 368 and trained our weight sparse OpenPose CNN using the COCO training dataset consisting of over 100K person instances labeled with over one million total body parts. Table 3.1 compares the weights for both the baseline CNN and the proposed sparse one. As shown in Table 3.1, the baseline CNN requires 14,808 18Kb BRAMs, which exceeds the modern FPGA resources. On the other hand, the sparse weight CNN achieved more than 90% of the weights. Thus, the necessary number of BRAM is only 933.

3.5.2 Implementation Results

We implemented the proposed sparse weight OpenPose on the Xilinx Inc. Xilinx Virtex UltraScale+ FPGA VCU1525 acceleration development kit with a host PC. We used a 16-bit fixed-point precision weight for the CNN implementation. The FPGA board has a Xilinx Inc. Virtex Ultrascale+ XCVU9P FPGA, which has 788 K LUTs, 1,576 K FFs, 4,320 18Kb BRAMs, 960 UltraRAMs, and 6,840 DSP slices. We used the Xilinx Inc. SDAccel 2018.2 with a timing constraint of 300 MHz. Our implementation used 213,225 LUTs, 124,280 FFs, 2,510 18Kb BRAMs, 480 UltraRAMs, and 1,520 DSP48Es.

Table 3.1: Comparison of the Number of Weights.

	Baseline CNN		Sparse Weight CNN		
Layer	#Weights	#BRAMs	#Weights	ratio(%)	#BRAMs
conv1_1	1728	2	1058	61.2	1
conv1_2	36864	32	3133	8.5	3
conv2_1	73728	64	3339	4.5	3
conv2_2	147456	128	5750	3.9	5
conv3_1	294912	256	10911	3.7	10
conv3_2	589824	512	21233	3.6	19
conv3_3	589824	512	20962	3.6	19
conv3_4	589824	512	20830	3.5	19
conv4_1	1179648	1024	100270	8.5	88
conv4_2	2359296	2048	200540	8.5	175
conv4_3	1179648	1024	100270	8.5	88
conv4_4	294912	256	25067	8.5	22
Stage1_conv1_B1	147456	128	8779	6.0	8
Stage1_conv2_B1	147456	128	10995	7.5	10
Stage1_conv3_B1	147456	128	9696	6.6	9
Stage1_conv4_B1	16384	15	786	4.8	1
Stage1_conv5_B1	4864	5	1084	22.3	1
Stage1_conv1_B2	147456	128	9546	6.5	9
Stage1_conv2_B2	147456	128	10007	6.8	9
Stage1_conv3_B2	147456	128	16325	11.1	15
Stage1_conv4_B2	16384	15	2375	14.5	3
Stage1_conv5_B2	4864	3	323	13.3	1
Stage2_conv1_B1	1160320	1008	39153	3.4	34
Stage2_conv2_B1	802816	697	34725	4.3	31
Stage2_conv3_B1	802816	697	46866	5.8	41
Stage2_conv4_B1	802816	697	57779	7.2	51
Stage2_conv5_B1	802816	697	81174	10.1	71
Stage2_conv6_B1	16384	15	3709	22.6	4
Stage2_conv7_B1	4864	5	253	5.2	1
Stage2_conv1_B2	1160320	1008	43615	3.8	38
Stage2_conv2_B2	802816	697	29021	3.6	26
Stage2_conv3_B2	802816	697	28406	3.5	25
Stage2_conv4_B2	802816	697	40249	5.0	35
Stage2_conv5_B2	802816	697	57955	7.2	51
Stage2_conv6_B2	16384	15	4541	27.7	4
Stage2_conv7_B2	4864	5	2656	54.6	3
Total	17048128	14808	1053381	6.1	933

Also, it satisfied the timing constraint for real-time applications. Since our architecture computed an image with 42.6 ms, the number of frames per second (FPS) was 23.43. We measured the total board power consumption: It was 55 W. Thus, the performance per power efficiency was 0.444 (FPS/W).

3.5.3 Compared with a High-End GPU

We compared our FPGA-based OpenPose with a high-end GPU. We used the NVidia Titan X Pascal architecture GPU. Also, we measured the total power consumption. Note that, in the experiment, we set the number of batch sizes to one to measure the latency. Its number of FPS was 6.7 on average, and power consumption was 195 W. Thus, the performance per power efficiency was 0.034 (FPS/W). Therefore, the FPGA was 3.49 times faster, dissipated 3.54 times lower power, and its performance per power efficiency was 13.05 times better.

3.6 Conclusion

We implemented the OpenPose, a deep learning-based pose estimator on the Xilinx Inc. Virtex Ultra-Scale+ FPGA VCU1525 acceleration development kit with a host PC. We introduced a sparse weight CNN to reduce the memory size for weights, which is dominant in the memory size. Then, we proposed the indirect memory access architecture to efficiently realize the sparse CNN convolutional operation. Also, to increase throughput further, we applied the six stages of pipeline architecture with a feature map memory as a pipeline buffer. Our implementation satisfied the timing constraint for real-time applications. Since our architecture computed an image with 42.6 ms, the number of frames per second (FPS) was 23.43. We measured the total board power consumption: It was 55 W. Thus, the performance per power efficiency was 0.444 (FPS/W). Compared with the NVidia Titan X Pascal architecture GPU, it was 3.49 times faster, it dissipated 3.54 times lower power, and its performance per power efficiency was 13.05 times better.

Chapter 4

Weight Sparseness for a Feature Map Split-CNN Toward Low-End FPGAs

4.1 Introduction

4.1.1 Computational Bottleneck on Low-End FPGAs

When implementing CNNs on low-end FPGAs, the bottleneck is the amount of memory required to store the CNN feature maps. For example, a VGG [10] model with an input resolution of 224×224 has a feature map of $224 \times 224 \times 64$ pixels in at maximum. If each pixel represents 8 bits (1 byte), then the data quantity of this feature map will be 3.06 MB. Since the calculation requires memory to store input and output data, the total future maps are 6.12 MB. Low-end FPGAs, such as the PYNQ-Z1 FPGA board, have only 630 KB of on-chip memory and cannot store the entire feature map. Therefore, external memory must be used, but the external memory bandwidth may become a bottleneck.

We use the VGG16 model with 90% of the weights sparseness ratio. The sum of the input and output feature map of the second layer, which is a significant bandwidth bottleneck, is 6.12 MB, and the weights are 3.6 KB. The computation required is 0.17 GMAC (Giga Multiply-Accumulate). With Double MAC [44], one DSP can perform two MAC operations in one cycle, and if the 220 DSPs on the PYNQ-Z1 FPGA board are running at 200 MHz, then the amount of computation per second is 88 GMAC. The bandwidth of the external memory is approximately 1.7 GB/s. In this case, the calculation takes about 1.9 ms to complete, while the transfer of the feature maps takes approximately 3.6 ms. Therefore, we can see that the bandwidth with the

external memory is the bottleneck. Although the ratio of weights becomes larger in the other layers of VGG, the required bandwidth does not exceed the bandwidth of the external memory, so the weights do not become a bottleneck even if they are read from the external memory.

4.1.2 CNN Implementation on Low-End FPGAs

We implement in a low-end FPGA with more restrictions on on-chip memory resources and external memory bandwidth. Existing methods are limited by communication with external memory. Because the FPGA used in an embedded system has a narrow bandwidth with external memory, storing data in on-chip memory to realize high speed processing is necessary. The on-chip memory in a low-end FPGA is small; hence, it cannot store an entire feature map. Therefore, it is imperative to rely on external memory for buffering.

We employ a split convolutional neural network (split-CNN) [45] to solve the problem of external memory bandwidth. An input image is spatially split into small patches, and each patch is processed individually. A patch in which the spatial resolution is reduced by splitting can be inferred as a smaller memory footprint. Splitting does not consider the data dependencies between layers. Thus, it is possible to design implementations that efficiently use small, fast-access on-chip memories.

We propose a scheduling method on FPGAs with small-footprint memory using split-CNN. In split-CNN, the input image is spatially split into small patches, and each patch is processed individually. We process the patches in time-division processing and reuse the buffer. Since the intermediate data of a typical CNN is larger than the on-chip memory of a low-end FPGA, it needs to be buffered in off-chip memory. We can store all the data in on-chip memory by using the scheduling, even for large resolution inputs. We also propose an FPGA implementation that accelerates sparse convolution by storing all intermediate data in on-chip memory. All intermediate data are stored in on-chip memory according to the proposed scheduling. We achieved high speed sparse convolution with high computational efficiency and no memory bandwidth bottleneck.

4.2 Memory Size Optimizations

4.2.1 Definition of a Split-CNN

Although on-chip memory in an FPGA is fast, implementing CNN in low-end FPGAs is hindered by their limited memory size. Meanwhile, a CNN requires significant buffering to retain a feature map. Using large off-chip memory, such as DDR4 and HBM2, addresses the memory size limitation problem. However, an external memory bandwidth limitation results in a performance bottleneck. The resolution of the feature map is large in the first half of the CNN and small enough to be stored in the on-chip memory of FPGA in the second half. Therefore, by reducing the feature map size only for the first half of the CNN, all intermediate data can be stored in on-chip memory, eliminating the communication bottleneck with external memory. By splitting, the size of the feature map in the step before concatenate becomes $1/k$ of the size without splitting so that the entire feature map can be buffered only in on-chip memory. In other words, it works like cache blocking. As a result, access to external memory for intermediate data can be eliminated.

To address the limitations of external memory bandwidth and its size, we employ a split-CNN [45] that splits an input image into small spatial patches and tests each patch using a CNN model. Each patch independently collapsed in the first half. Subsequently, patch concatenation is performed, and the merged feature map is applied to the final layer. Although this splitting does not consider the data dependence between the shallow layers, the entire layer is used by the subsequent layers after concatenation. Accuracy may scarcely deteriorate or improve because of the split. We demonstrated a slight accuracy drop through experiments.

4.2.2 Operation of a Split-CNN

Fig. 4.1 illustrates an example of a split-CNN operation. A five-layer CNN uses four sub-images. The cube represents the feature map size. A quadrangular pyramid between the cubes indicates layer operations, such as convolution and pooling.

A split-CNN splits the primary input image into smaller spatial patches (four in this example) and performs inference operations on each patch. Because the pooling operation reduces the feature-map size and becomes smaller in the subsequent layers, an on-chip memory can store the feature map. After several CNN layers are inferred, the feature maps are concatenated. Following the merging of the temporary feature maps, a convolutional operation is performed on the entire feature map region to obtain a detection result with

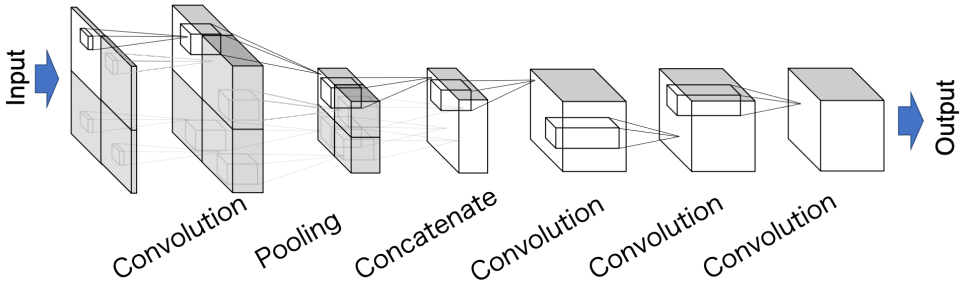


Figure 4.1: Example of operation for split-CNN, which splits into four patches.

dependency on the whole input image.

4.2.3 Scheduling of Computation

The feature map represented by the white cube in Fig. 4.1 indicates the memory size. For feature maps represented as black-filled cubes, we reuse the on-chip buffers. Therefore, memory resource allocation for the entire feature map is not essential.

The computation scheduling, including memory allocation and its access to hardware implementation, is as follows: The input image is split into patch inputs for the CNN. Two convolutional operations of the first two layers and the pooling operations are performed for each patch. The outputs are stored in the scratch-pad memory implemented in BRAM prepared for concatenation. The remaining three patch images are analogously examined and stored in the scratch-pad memory. The four patched images are merged into a single feature map. After completion, a merged feature map can have the same size as a conventional CNN. The latter causes resolution reduction. If concatenation is performed immediately after pooling, the scratch-pad memory is minimized. We sequentially consider a patched image; hence, preparing a small feature map memory for a patch image is sufficient. The number of operations and the weight (parameter) size are the same as those of a conventional CNN that processes an entire image. The merge operation has a computational overhead; it requires additional data movement for feature map merging, but it is negligible because it is an on-chip memory operation.

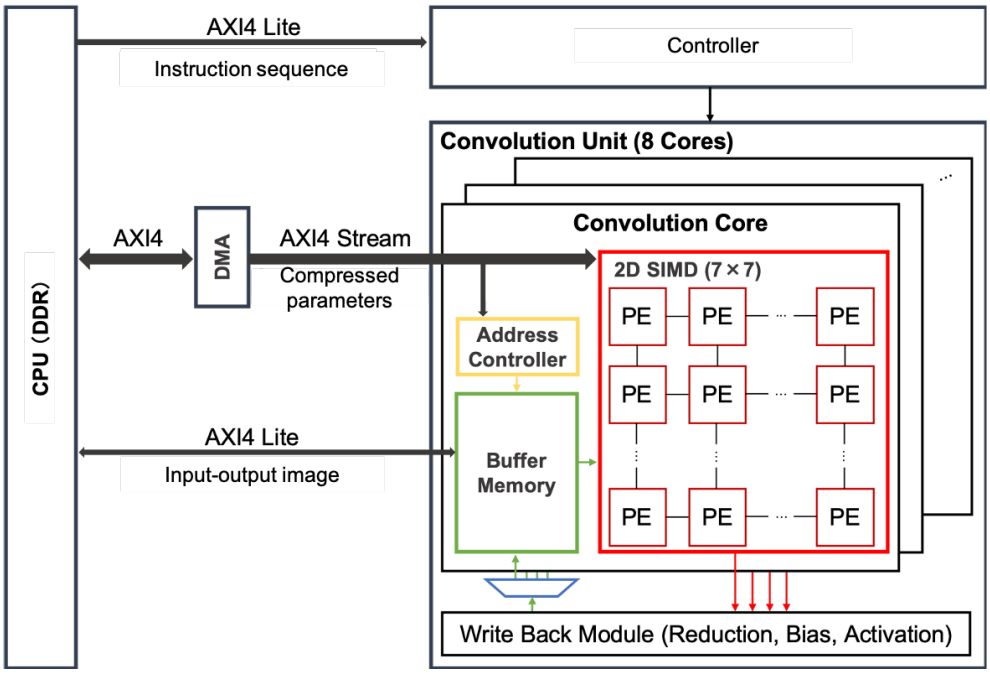


Figure 4.2: Overall architecture.

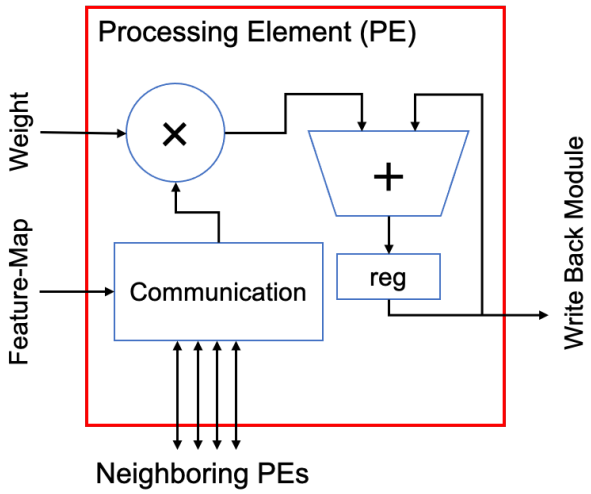


Figure 4.3: Processing element.

4.3 FPGA Implementation

4.3.1 Overall Architecture

Fig. 4.2 shows the overall architecture. It consists of a convolution unit that performs a convolution operation, controller, and direct memory access (DMA) controller. The control signals for the convolution and concatenation operations are offloaded to the FPGA circuit. The host CPU performs image transfer, image preprocessing, FPGA control, and calculation of a neural network of a fully connected layer. In our implementation, the I/O data are transferred asynchronously using the AXI4Lite protocol. Generally, it is used for parameter setting and circuit control owing to its narrow bandwidth. However, because our circuit contains only the input and output of the input image and the output of the inference result, the bandwidth for AXI4Lite is sufficient for real-time performance. Additionally, AXI4Lite can be accessed asynchronously from a host program, facilitating communication hiding and debugging during development and the AXI4Stream protocol. Meanwhile, the compressed sparse weight parameter and weight index are burst transferred by the AXI4Stream protocol via the DMA controller.

The convolution unit includes several convolution cores and a write-back module. The number of cores depends on available FPGA resources. Each convolution core consists of an address controller, buffer memory, and 2D SIMD connected to a two-dimensional grid. The 2D processing elements (PEs) calculate the vertical and horizontal images of the input image in parallel. If the resolution is a constant resolution multiple, then all PEs operate efficiently without a fraction. Because the VGG model uses an input resolution of 224×224, we set the size of the PEs to 7×7. Each PE is responsible for the input image area and has spatial computational parallelism relative to the input feature map. Each core stores a distributed set of feature map data divided in the channel direction. Therefore, the number of cores indicates the computational parallelism of the number of input channels.

The write-back module returns the output feature map generated by the 2D SIMD operator to the buffer memory. Each convolution core is assigned in the channel direction of the input image because the output computed for each core needs to be written back across the cores. The output core data are reduced, and the addition of a bias and processing of an activation function is performed. Two types of activation functions are implemented: $ReLU$ and *Linear*. If the reduction calculation is completed, the output destination core is selected by the demultiplexer, and the data are written back to the buffer memory.

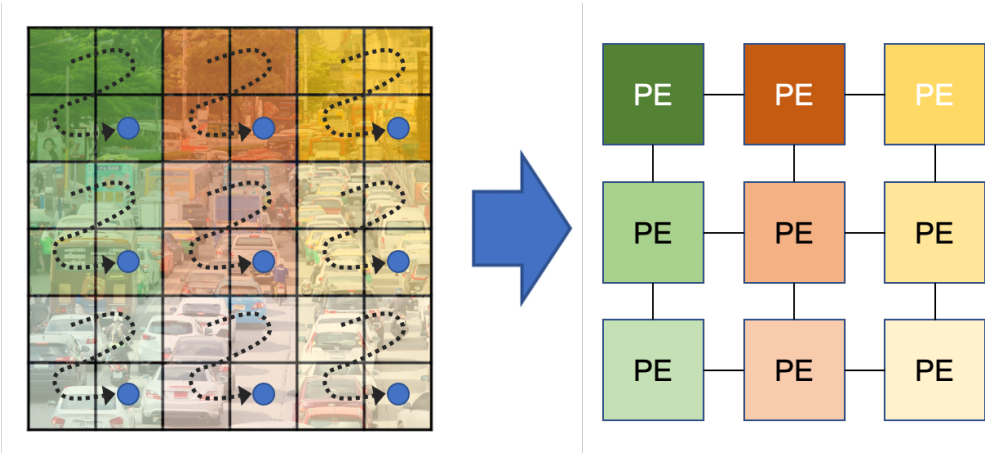


Figure 4.4: Assignment of feature maps into 2D PEs.

4.3.2 Processing Element Architecture

Fig. 4.3 shows the processing element (PE) structure consisting of a multiplier, an accumulator, and a communication unit. Each accumulator stores one output pixel data. The weight and feature map are sent as input signals. The weight value is the stream transferred from external memory to all PEs in a certain core. The wide-band feature map loaded from the buffer memory is transferred in a one-to-one ratio to each PE for the feature map. Therefore, buffer memory has a word width of the cores $\times$ feature map bit width.

The PE communication unit is connected to the neighboring PEs at the top, bottom, left, and right. Convolution requires the data of adjacent pixels owing to the stride effect. Two three-input selectors are implemented for the up-down selection and left-right selections. If the two selectors are enabled simultaneously, communication is performed with PE in the oblique direction. All PE selectors are synchronized.

We represent the weight and feature map by 8-bit fixed-point numbers and multiplier treatment by an 8-bit input / 16-bit output. The accumulator register has 16-bit precision. Multipliers and accumulators are implemented using digital signal processing (DSP) blocks. A double MAC operation technique [44] computes two MAC operations simultaneously with one DSP block per clock. Because the available number of bits for the DSP is 25 $\times$ 18, it computes $y = (x\_1 << 9 + x\_2) \times w$ and then obtains $y\_1 = (y >> 9)$ & $mask$ and $y\_2 = y$ & $mask$. Therefore, the number of PEs up to twice the number of DSP blocks of the FPGA resource can be implemented by the DSP blocks.

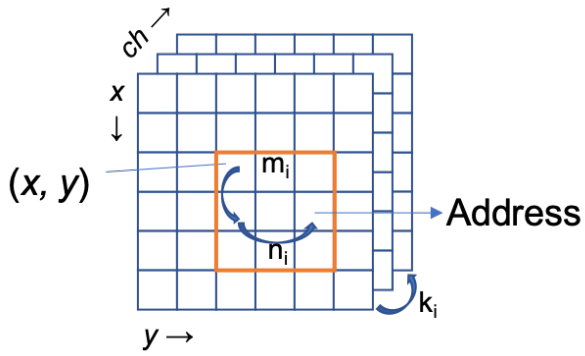


Figure 4.5: Example of buffer memory addressing.

4.3.3 Operation and Communication for PEs

In the proposed circuit, all feature maps are stored in the on-chip buffer memory in the PE. It is possible to perform sparse convolution with random access at high speed. Fig. 4.4 shows a typical feature map allocation to the PEs. We assume that the feature map of size 6×6 is assigned to the 3×3 PEs. Each PE is set spatially continuous feature map data. We carefully assign them so that feature map data are not duplicated. If the feature map size is not a constant PE multiple, a zero padding is performed. If the input and output feature maps are the same size, then the coordinates and PEs have a one-to-one correspondence.

The communication between the PEs is as follows: In a convolutional operation, neighboring data in the input feature map must obtain a certain coordinate output. The required neighboring data range is the same as the convolution kernel size. Spatially continuous feature maps are assigned to the same PE. Therefore, the required input feature map is stored in the same PE as the output feature map in many cases. If the output feature map is the area boundary allocated to the PE, then it is necessary to obtain the buffer memory data of the adjacent PE. Communication between PEs occurs if the dependence of the calculation straddles the PE assignment boundary. Two three-input selectors are used for up-down and left-right selections. Communication overhead can be concealed using a pipelined selector.

4.3.4 Zero-Skip Computation for Sparse Weights

We explain the format of the compressed weights and the method for realizing zero skipping. We assume that almost all the weights have zeros. Therefore, the trained weights can be compressed by removing zeros. Compressing the

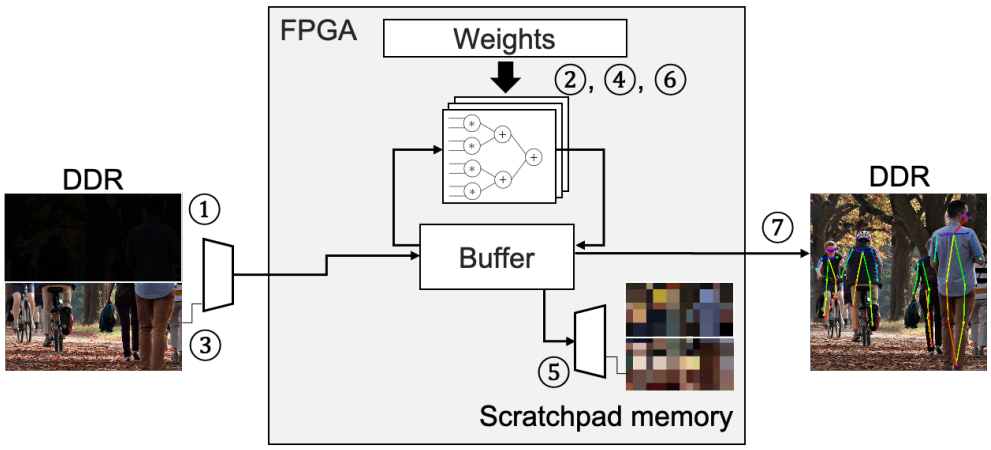


Figure 4.6: Example of scheduling for splitting into two patches.

weight reduces the memory size and bandwidth. To retain skipped zero-weights during convolution, it is necessary to store a location for zero elements. Therefore, we created a pair of non-zero weight values and indices and stored them in the off-chip memory. Because we train the CNN model with a high sparse ratio (typically more than 90%), the overhead of the index memory size is low.

The weighted index comprises the m, n, and k coordinate of the kernel. The coordinates of the kernel indicate the original address of the convolution kernel. Here, m and n denote the vertical and horizontal coordinates, respectively, and k denotes the input feature map channels.

Fig. 4.5 shows an example of a buffer memory address. Zero-weight restoration is performed in the address controller. The address of the buffer memory is computed by The following formula computes the address of the buffer memory:

$$Address \quad = \quad Sx \times (x + m_i) + Sy \times (y + n_i) + Sch \times k_i$$

where $Address$ is the buffer memory address, and Sx, Sy, and Sch indicate the vertical, horizontal, and channel directions of the memory offset, respectively. Further, x and y are the coordinates of the input image, and m_i, n_i, and k_i are the weight indexes.

Because a different weight index is transferred to each convolutional core, the buffer memory address differs. A pair of weight data and its index are burst-transferred to each convolution core during convolution. It enables the performance of a MAC operation with the weight data in each cycle without the PE in an idle state.

Table 4.1: Split and feature map sizes in CIFAR-10 dataset.

#Splits (k)	Feature-map size	Error	
None	64.0 KB	6.4%	
2	32.0 KB	6.6%	(+0.2)
4	16.0 KB	7.3%	(+0.9)
8	8.0 KB	9.0%	(+2.6)
16	4.0 KB	12.9%	(+6.5)
32	2.0 KB	23.9%	(+17.5)
64	1.0 KB	28.0%	(+21.6)

Table 4.2: Split and feature map sizes in CIFAR-100 by dataset.

#Splits (k)	Feature-map size	Error	
None	64.0 KB	29.7%	
2	32.0 KB	30.1%	(+0.4)
4	16.0 KB	30.5%	(+0.8)
8	8.0 KB	34.7%	(+4.7)
16	4.0 KB	44.1%	(+14.4)
32	2.0 KB	56.6%	(+26.9)
64	1.0 KB	58.0%	(+28.3)

4.3.5 Scheduling for Feature Map Split

Fig. 4.6 shows an example of scheduling for two split images. The image is split on the host CPU, and the upper half is transferred (Fig. 4.6 (1)). Convolution operations are performed on the transferred image in several layers in the first half of the CNN, and the result is stored in the scratch-pad memory (Fig. 4.6 (2)). Furthermore, the lower half of the image is transferred, computed, and stored in the scratch-pad memory (Fig. 4.6 (3) and (4)). The sub-feature maps are concatenated (Fig. 4.6 (5)). Besides, the concatenated feature map is written back to the buffer memory, and the latter half of the CNN layer is operated (Fig. 4.6 (6)). All layer evaluations are completed and transferred to the DDR memory (Fig. 4.6 (7)). Finally, the procedure for inferring one image is completed.

Table 4.3: Split and feature map sizes in ImageNet by dataset.

#Splits (k)	Feature-map size	Error	
None	3.06 MB	27.6%	
2	1.53 MB	27.8%	(+0.2)
4	0.77 MB	28.0%	(+0.4)
8	0.38 MB	28.1%	(+0.5)
16	0.19 MB	29.9%	(+2.3)
32	0.10 MB	31.3%	(+3.7)
64	0.05 MB	37.9%	(+10.3)

Table 4.4: FPGA implementation results.

	LUT	FF	DSP	18K BRAM
Consumption	31,465	26,692	204	258
Utilization	59.1%	25.1%	92.7%	92.1%

Table 4.5: performance for each convolutional layer.

Size	Sparse	Latency	Measured Perf.	Peak Perf.	Efficient
56×56	0%	24.4 ms	151.6 GOP/s	156.8 GOP/s	96.7%
56×56	80%	6.0 ms	613.5 GOP/s	784.0 GOP/s	78.3%
56×56	85%	4.8 ms	769.1 GOP/s	1,045.3 GOP/s	73.6%
56×56	90%	3.6 ms	1,033.3 GOP/s	1,568.0 GOP/s	65.9%
56×56	95%	2.3 ms	1,601.5 GOP/s	3,136.0 GOP/s	51.1%
56×56	80%	6.0 ms	613.5 GOP/s	784.0 GOP/s	78.3%
28×28	80%	1.6 ms	567.3 GOP/s	784.0 GOP/s	72.4%
14×14	80%	0.5 ms	444.6 GOP/s	784.0 GOP/s	56.4%

Table 4.6: Comparison of the performance of VGG16 classifications.

	GPU	ASIC[46]	FPGA[47]
Platform	Jetson Nano	Edge TPU	Zynq-7020
Throughput	10.0 FPS	2.8 FPS	5.7 FPS
Performance	393 GOP/s	110 GOP/s	-
Clock Freq	850 MHz	-	214 MHz
Power	10 W	2 W	3 W
Precision	FP32	INT8	INT8
Accuracy	71.8%	-	67.7%

	FPGA[48]	FPGA[49]	**Ours**
Platform	Stratix V	Xilinx VC709	Zynq-7020
Throughput	7.0 FPS	-	30.8 FPS
Performance	-	1,713 GOP/s	1,210 GOP/s
Clock Freq	200 MHz	200 MHz	200 MHz
Power	8 W	-	3 W
Precision	INT32	INT8	INT8
Accuracy	-	-	67.8%

4.4 Experimental Results

4.4.1 Implementation Setup

Our environment was PyTorch 1.7.1, Ubuntu 20.04, and RTX 3090. We designed the proposed CNN architecture using the Xilinx Inc. Vivado HLS 2020.1 and implemented it on a Digilent Inc. PYNQ-Z1 FPGA board (FPGA: Xilinx Inc. Zynq XC7Z020). The host program was written in Xilinx Inc. PYNQ 2.5, whose library controls the FPGA circuits in Python.

4.4.2 Buffer Size and Accuracy by a Split-CNN

The CIFAR dataset includes ten classes for CIFAR-10 and 100 classes for CIFAR-100. Each dataset contains a total of 60,000 labeled images. We used the VGG16 CNN as a target model and increased the k from 2 to 64 splits. Each model was independently fully trained, and performed the test was conducted independently. The bit precision during training is a 32-bit float, and the GPU performs all calculations. The feature map capacity indicates the maximum feature map size at the time of 8-bit quantization.

Table 4.1 and Table 4.2 show the splits, maximum feature map size, and classification error rate. Each patch is processed independently, and neural connections in the network across the split boundary are ignored.

It will degrade the representative capability of the CNN model and thus degrade the recognition accuracy. Therefore, there is a trade-off between the feature map size and classification accuracy corresponding to the splits. The misclassification rate was increased with increasing splitting in both datasets from the experiments.

The ImageNet dataset contains approximately 1 million images with 1000 labeled classes. It is a practical classification task benchmark with sufficiently large image resolution. We used the VGG16 CNN as the target model from 2 to 16 splits. Each model is independently trained with a full scratch and the same hyperparameters as in the CIFAR datasets. The feature map capacity denotes the maximum feature map size using 8-bit quantization.

Table 4.3 shows the number of splits k, the maximum feature map capacity, and the classification error rate. Similar results were obtained for the CIFAR dataset.

4.4.3 Resource Usage and Peak Performance

Because the buffer memory of each convolutional layer is implemented by a block RAM (BRAM), if the number of splits k increases, the BRAM usage decreases. The BRAM usage for the buffer memory was reduced by 85% in 16 splits than the no split case and fit in the PYNQ-Z1 FPGA board resource. The proposed method reduces the buffer memory usage toward a low-end FPGA. The two splits did not reduce the BRAM usage due to the scratch-pad memory overhead for concatenation.

Table 4.4 shows the actual hardware consumption after placement and routing. We have designed an architecture for the VGG16 model with 16 splittings. Since sparsity is sourced from off-chip memory as a weighting parameter, the same architecture is used for all sparsity. The DSP and BRAM consumed more than 90%; thus, the proposed circuit effectively utilizes the available FPGA resources. The proposed architecture contains eight cores and 392 PEs in total. The clock frequency was 200 MHz; thus, the peak performance of non-spars was 156.8 Giga operations per second (GOP/s).

4.4.4 Performance Analysis for Each Layer

We analyzed the relationship between the sparse ratio and system performance. We compared the theoretical performance with the measured performance to examine the computational efficiency of the proposed architecture. We investigated how the performance and the operation unit usage change based on practical layer parameters. The proposed architecture stores all intermediate data in on-chip memory. Hence, we do not consider the transfer

between the host processor and the FPGA.

Table 4.5 presents the throughput of each layer. If the sparse ratio is high, then the processing of the MAC operation is reduced; thus, the write-back of the computation result becomes relatively a bottleneck. In the model with the sparse ratio of 80%, the computational efficiency exceeded 70%, and it is observed that the model had high computational efficiency. In other words, almost all MAC units were used for the computation. If the input resolution is small, then the weights transfer becomes relatively overhead, resulting in lower computational efficiency of the operation units. For an FPGA with small on-chip memory, storing weight data in a large-capacity external memory is necessary. Therefore, it is challenging to improve weight transfer time further.

4.4.5 Comparing Classification Performance

For an image classification task using ImageNet, the inference speed and power efficiency were measured using the proposed circuit. The model used was VGG16 for a 224×224 input image for ImageNet 2012 dataset. An 8-bit integer was used for weights and feature maps. In our model, the number of feature maps split is 16, and the weight sparsity is 90%. For comparison, we used the NVIDIA Jetson Nano for embedded GPU board and Google Edge TPU board for application-specific integrated circuit (ASIC). The weights and bit precision of the feature map to be compared are shown in the table. All platforms implemented VGG16. Only our FPGA implementation applied the weight sparseness techniques, with a 90% ratio. To train our sparse weight for a split-CNN, we use a gradual sparseness technique that can be easily applied across different sparseness ratios for several steps [50]. We used known training hyperparameters (learning ratio, number of epochs, initial weight values, and an optimizer) [51].

Table 4.6 shows a comparison of the throughput and power efficiency for GPU, ASIC, existing FPGA, and our FPGA implementation. From the experiment, our implementation was 3.1 times faster than the GPU, 11.0 times faster than the ASIC, and 5.4 times faster than an existing implementation using the same FPGA. Our implementation was superior to all platforms in terms of power efficiency. The accuracy has been reduced by sparseness and feature map splitting but was still higher than the existing same FPGA implementation [47]. In addition, our implementation is faster and less power consumption than another low-end embedded FPGA implementation [48] and achieved the same speed as the FPGA implementation [49] with ten times more resources than ours.

4.5 Conclusion

This study proposed an architecture and scheduling method using a sparse weight split-CNN for a low-end FPGA for an embedded vision system. Our implementation resolved the memory bottleneck on low-end FPGAs by splitting the feature maps and scheduling on-chip memory buffering. Also, we developed an architecture for CNNs with high computational efficiency for sparse weights. Our architecture achieves high speed by storing the internal data in on-chip memory. We implemented the proposed architecture on the PYNQ-Z1 FPGA board, a low-end FPGA. The experiment on classification using VGG16 with ImageNet 2012 dataset shows that our implementation was 3.1 times faster than the GPU, 11.0 times faster than the ASIC, and 5.4 times faster than an existing FPGA implementation.

Chapter 5

An FPGA Implementation of a Multi-Core Neural Network Design using HLS

5.1 Introduction

5.1.1 FPGA Design using HLS

Designing FPGAs at the logic circuit level using hardware description language (HDL) is much more time-consuming than the software-based design of CPUs and GPUs. In recent years, attention has been focused on High-Level Syntheses (HLS), which can synthesize logic circuits from the C programming language. High-Level Syntheses (HLS) generates HDL based on programs written in high-level languages such as C to realize logic circuits. Conventional HDL design requires a specific behavioral design at the clock cycle level, which is time-consuming and difficult to verify. HLS design enables design at a high level of abstraction using languages such as C suitable for algorithm description, thus improving design efficiency. Existing test tools for high-level languages can be used to verify the operation. It is also possible to change resource and clock constraints by configuration. By inserting the pragma directive, pipelined and parallelized computational circuits standards-compliant input/output and control circuits can be easily generated. Xilinx's Vitis HLS [52], Intel's Intel FPGA SDK for OpenCL [2], and others have been put to practical use.

5.1.2 Limitations of HLS Design

HLS design can be realized in a relatively short time. For example, Yang et al.'s study [3], which proposes an FPGA implementation using HLS, was implemented by two workers in one month. On the other hand, there is a large difference in the achievable operating frequency between the HLS-generated design and the manual design using HDL. Because it is difficult to estimate the interconnect delay at the HLS level accurately. This problem is known to be even more pronounced when implementing large HLS designs [4].

In this chapter, we propose a multi-core architecture for fast CNN inference. A high operating frequency can be achieved even in large HLS-based FPGA designs by dividing a large HLS kernel into multiple small cores.

5.1.3 Parallelism in CNN

The output feature map of each layer of CNN is a set of independent output pixels computed by the inner product of the input feature maps and weight. When considering the speedup in computing each output pixel, it is necessary to copy and distribute the input data to each output if the output pixels are parallelized. It is necessary to aggregate the partial sums due to the computational dependency if the input pixels are parallelized. In this chapter, we propose a method to solve the problem of input feature map dependency and perform parallel computation without using complex structures by communicating partial sums in a unidirectional ring. The increase in latency caused by the communication is masked by the time-division processing using the independent nature of the output feature maps. It is possible to reduce the degree of coupling between the computations of each feature map By parallelizing the computation in this way. This has the advantage of increasing the operating frequency because the control circuit can be had on each core, shortening the circuit's synthesis time.

5.2 Architecture

5.2.1 Overall Architecture

In this chapter, we propose an architecture for a fast CNN accelerator called the *Wasabi Engine*. The block diagram in Fig. 5.1 shows the overall picture of the *Wasabi Engine*. The *Wasabi Engine* is an architecture consisting of multiple cores. Each core has independent registers and executes its own sequence of instructions. In addition to the cores, the *Wasabi Engine* consists of FIFOs that connect the cores and Direct Memory Access (DMA)

circuits that communicate with external memory. Each core is connected by a unidirectional ring bus through the FIFO, and the number of cores can be configured according to the resources. Each core is connected to a DMA circuit for supplying instruction strings from external memory. Some cores are also connected to DMA circuits to communicate feature map data with external memory.

Vitis HLS designs each core of the *Wasabi Engine*. Each core is separated by a simple FIFO connection, which facilitates optimization by EDA tools. HLS generates the cores as a small design, and multiple cores are designed by connecting them to achieve a high operating frequency. The FIFOs and DMAs that connect the cores are IPs provided by Xilinx. The connection between the designed cores and IPs was designed using the Xilinx Vivado IP integrator.

5.2.2 Primary Interface

The *Wasabi Engine* has an instruction stream and a data input/output stream as interfaces. The instruction stream is an AXI Stream standard stream that transfers 32-bit instruction strings. Each core has its independent instruction stream, and instruction strings are supplied to each core from external memory by DMA. The data input/output stream is an AXI Stream standard stream that transfers 8-bit feature map data. Only one data input stream and one data output stream are implemented in the entire circuit, and they are connected to a specific core. Data input and output to the cores not connected to the data input/output stream is realized by communication via the ring bus between the cores.

5.2.3 Core Architecture

A block diagram of the core architecture is shown in Fig. 5.2. The core consists of the instruction cache, instruction controller, SIMD unit, buffer memory, ALU, and ring bus. The core is designed in the C language using HLS, and it pipelines the circuits for parallel operation.

The instruction cache is a direct-mapped read-only cache for instruction strings. If a cache miss occurs, the DMA circuit reads data from external memory. The instruction controller performs instruction decoding, instruction execution, and circuit control. It has a program counter, general-purpose registers, adders, comparators, etc., and performs state transitions in loops. The controller sends the decoded control signals to other modules when executing a SIMD instruction. The SIMD unit consists of an array of processing elements (PEs) connected in a two-dimensional pattern and mainly

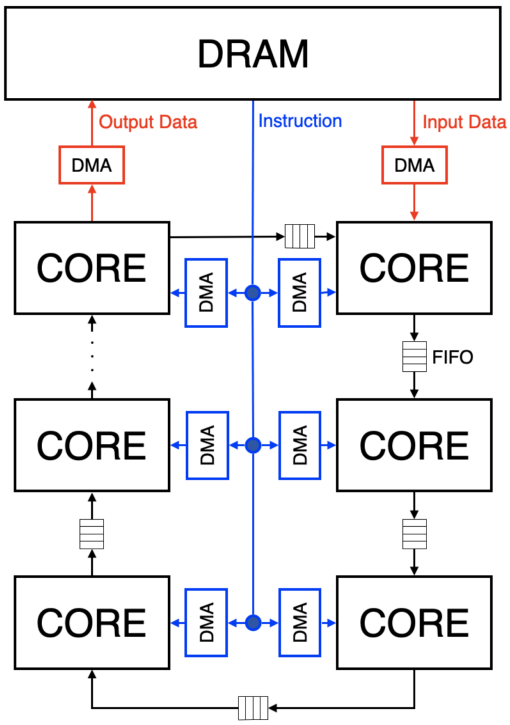


Figure 5.1: Overall architecture.

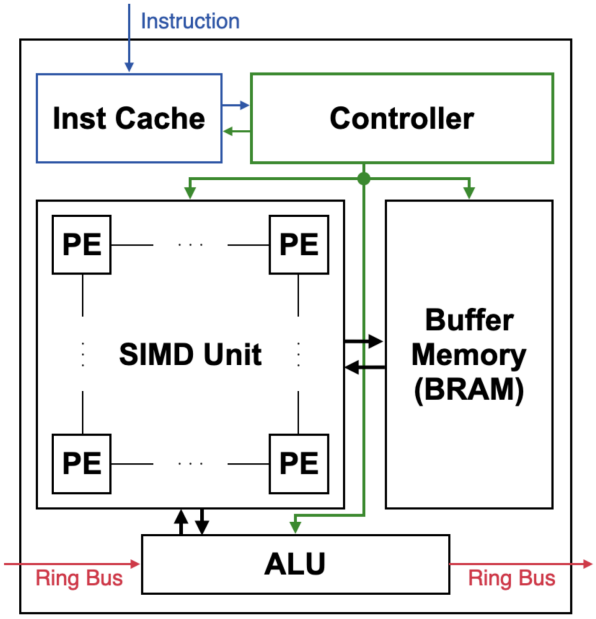


Figure 5.2: Architecture of the core.

performs the numerical computation of CNNs. The PEs are connected in a two-dimensional way to compute kernels that extend horizontally and vertically in the feature map. The buffer memory is an on-chip memory for storing the feature map and provides high bandwidth data to the SIMD unit. The data width of the buffer memory corresponds to the PE of the SIMD unit on a one-to-one basis. The arithmetic logic unit (ALU) is a circuit that performs the numerical computations on a single element and mainly performs operations on CNN layers, including contraction operations. The ALU is primarily composed of arithmetic circuits and is connected to a bus for one-to-one communication with each PE of the SIMD unit and a ring bus for communication with neighboring cores. The ring bus is for data communication with neighboring cores and is connected in a unidirectional ring. The SIMD unit and ALU registers can be sent to the adjacent cores. It is used to transfer the input/output feature map to/from the SIMD unit registers and to compute the data of other cores in the convolution operation.

5.2.4 Processing Element Architecture

The architecture of each PE is shown in Fig. 5.3. Each PE consists of a multiplier, an adder, an accumulator register, and an activation circuit. The activation circuit consists of an arithmetic right shift operator for downscal-

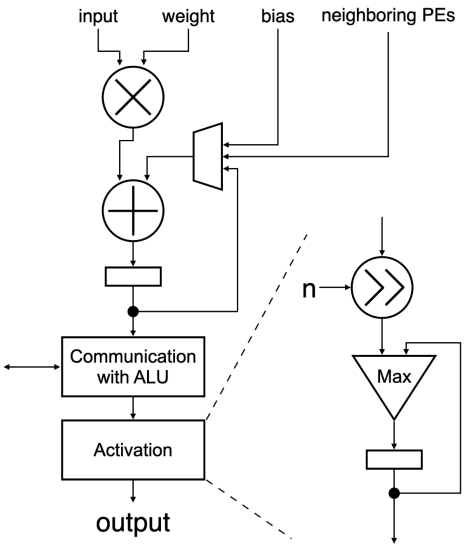


Figure 5.3: PE architecture.

ing, a comparator, and a register for comparison. The multiplier, adder, and accumulator register are used to perform the sum-of-products operation of convolution. When computing a CNN in integer representation, the result of the sum-of-products operation needs to be divided to prevent overflow. In this case, the right shift circuit can be used to perform the division by limiting the number of divisors to a power of two. The ReLU operation, which takes the maximum value of the target and zero, and the max pooling operation, which calculates the maximum value, are realized using a comparator and a comparison register. Each cycle, the PE is supplied with calculation data from the instruction controller and buffer memory, and the PE calculation results are written back to the buffer memory.

5.2.5 System Operation

First, we explain the allocation of the feature map. The feature map is three-dimensional tensor data, and each axis is referred to as the vertical, horizontal, and channel directions. The vertical and horizontal directions correspond to the coordinates of pixels in the input image. The channel direction is the size of the data that make up a single pixel for RGB color images, and the number of channels is 3. The feature map is divided into channel directions and assigned to different cores. In the case of an RGB color input image, each data is assigned to three cores. The feature map data assigned to different cores may be required for the Convolution calculation.

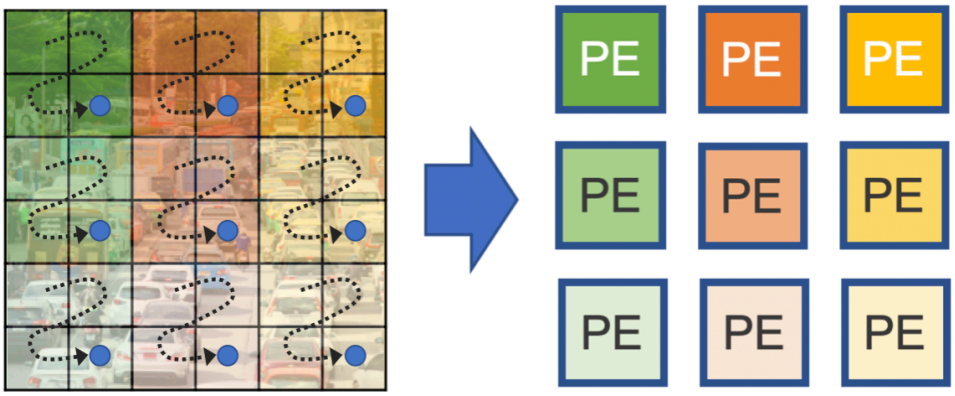


Figure 5.4: Allocation of feature map to PE array.

The cores are connected by a ring bus to access the feature map data assigned to other cores. Feature maps are divided horizontally and vertically within a core and assigned to PE arrays. The allocation of feature maps to PE arrays is shown in Fig. 5.4. The feature map is divided horizontally and vertically into adjacent regions and allocated to the BRAMs in the buffer memory corresponding to each PE. In the case of convolution computation, one PE is responsible for the computation corresponding to the output of one pixel. The entire output feature map can be computed by shifting the computation horizontally and vertically within PE arrays. The PE arrays are connected in a two-dimensional mesh, and the feature map data of neighboring PEs can be accessed.

Next, we describe the operation of the entire system during inference. First, the input image data prepared in the external memory is transferred to a core by the DMA circuit for input data. The transmitted data is then transferred to the corresponding core via the ring bus. After the transfer is completed, each core of the *Wasabi Engine* is supplied with a sequence of instructions by the DMA circuit and starts to compute CNNs. After all the CNN computation is completed, the feature map is transferred to the external memory by the ring bus and the DMA circuit for output data. When the transfer to the external memory is completed, the following input image is transferred.

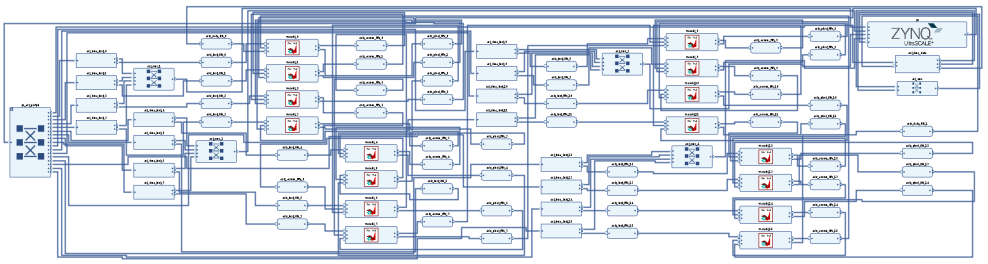


Figure 5.5: Block diagram of the Vivado IP integrator showing the connection between the 16-core kernel and the peripheral circuits.

5.3 Instruction Set

5.3.1 Overview

The *Wasabi Engine* has its instruction set architecture (ISA). The proposed ISA is a 32-bit RISC system. It has a dedicated SIMD instruction to compute CNNs. The immediate value of the instruction provides the parameters of CNN. The *Wasabi Engine* is a multi-core architecture, and each core executes one thread. The proposed ISA consists of control instructions, SIMD instructions, and compression instructions. The control instructions are used to set in the SIMD parameters and perform loop processing. SIMD instructions are used to perform MAC operations executed in SIMD by two-dimensional PE arrays. A compressed instruction is a SIMD instruction that packs four SIMD instructions by compressing SIMD instructions that satisfy specific conditions into eight bits and expressing them.

5.3.2 Control Instruction

The control instructions are mainly used to set the SIMD parameters and to perform loop processing. The control instructions include the SET, ADD, and BNE instructions. The SET instruction resets the values of general-purpose registers and parameter registers to their immediate values. The ADD instruction performs signed integer addition to general-purpose registers and is mainly used to calculate the counter of a loop. The BNE instruction is a conditional branch instruction that compares two general-purpose registers and branches when the values are different. This constraint and the cache mechanism allow the instruction sequence to perform burst transfers.

5.3.3 SIMD Instruction

SIMD instructions are instructions for performing numerical computations on SIMD units and ALUs. Most of the SIMD instructions consist of immediate values that are broadcast to each PE and instructions for the multiplexer of the SIMD circuit. Most SIMD instructions consist of an immediate value that is broadcast to each PE and a control part that expresses the control of the multiplexer of the SIMD circuit as a bit string. A single SIMD instruction can be used to A single SIMD instruction can perform address calculation, reading from the buffer memory, multiply-and-accumulate operations, ALU operations, maximum value operations, and writing back to the buffer memory in parallel. Since other operations such as address calculation are performed simultaneously as the sum-of-products operation, the utilization rate of the multiplier, which accounts for most of the computation time of the CNN, can be increased.

5.3.4 Compression Instruction

A compressed instruction is a type of SIMD instruction packing four SIMD instructions by expressing SIMD instructions that satisfy specific conditions in a compressed 8-bit format. By reducing the amount of data in the instruction sequence, it is possible to save bandwidth with external memory and the capacity of the instruction cache. Most of the SIMD instructions only perform address computation, call from memory, and perform product-and-accumulate operations, while other operations are partially executed. Therefore, the most commonly used SIMD instructions can be expressed in 8 bits. Up to four SIMD instructions can be compressed and stored in a single compressed instruction, executed in four-cycle steps.

5.4 Evaluation

5.4.1 High Frequency by Multi-Kernel HLS Design

We confirm that the operating frequency can be increased by multi-kernel design using HLS. The size of the PE array is 7×7, the number of cores is changed to 8, 16, and 20, and the circuit is implemented on a Xilinx FPGA board ZCU102. The circuit was designed using Vitis HLS 2021.1 and Vivado 2021.1. Table 5.1 shows the operating frequencies of the HLS single-kernel and multi-kernel designs and the LUT resource utilization of the multi-kernel design. In the single-kernel design, the operating frequency decreased when

Table 5.1: Operating frequency by design method and number of cores.

#Cores	Single	Multi	LUT
8	300 MHz	500 MHz	27 %
16	200 MHz	500 MHz	53 %
20	-	150 MHz	65 %

Table 5.2: Resource usage and utilization.

LUT	FF	36K BRAM	DSP
145.4k	300.0k	434	864
53.0 %	54.7 %	47.6 %	34.3 %

the circuit size was increased, and the Vitis HLS could not complete the HLS synthesis in the case of 20 cores.

On the other hand, the operating frequency did not decrease even when the circuit size was increased up to 16 cores in the multi-kernel design. In the multi-kernel design, the operating frequency decreased significantly when the number of cores was increased to 20. The experiments confirmed that the multi-kernel HLS design method is effective for large circuit sizes. To balance the number of cores and the operating frequency, we set the number of cores to 16 in the following experiments.

5.4.2 Implementation Results

We design the proposed architecture using Xilinx Vitis HLS 2021.1 and Vivado 2021.1 and implement it on Xilinx ZCU102. In the experiments in this chapter, the size of the PE array is 7×7, and the number of cores is 16. The block diagram of the Vivado IP integrator is shown in Fig. 5.5. Among the blocks in the figure, the block with the red figure is the 16 cores designed in the Vitis HLS. Table 5.2 shows the resource usage of the synthesis result. The LUTs and BRAMs utilize around 50%, while the DSP blocks have utilization of about 35%. The resource utilization of each module in the core circuit is shown in the Vitis HLS synthesis report. Table 5.3 shows the breakdown of the estimated resource usage by module in each core. We find that the instruction controller occupies the LUT, the SIMD unit occupies the DSP block, and the buffer memory occupies the BRAM.

Table 5.3: Estimation of resources by module within a core.

Module	LUT	FF	BRAM	DSP
Inst Cache	0.4 %	0.2 %	15.4 %	0.0 %
Inst Control	51.5 %	89.4 %	0.0 %	9.3 %
SIMD Unit	35.5 %	0.0 %	0.0 %	90.7 %
ALU	12.6 %	10.4 %	0.0 %	0.0 %
Buffer	0.0 %	0.0 %	84.6 %	0.0 %

Table 5.4: Comparison between the proposed method and other platforms.

	CPU	GPU	MB2[53]	**Ours**
Prec.	FP32	FP16	INT8	INT8
Freq.	2.4 GHz	1.4 GHz	0.3 GHz	0.5 GHz
Peak Perf.	0.3 TOPS	11.0 TOPS	1.4 TOPS	0.8 TOPS
Top1 Acc.	72.9 %	72.9 %	68.1 %	71.3 %
Power	-	19.9 W	-	32.7 W
Throughput	17.6 fps	52.8 fps	809.8 fps	123.5 fps
Latency	56.7 ms	18.9 ms	-	8.1 ms

5.4.3 Comparison with Other Platforms

We perform inference for MobileNetV2 using the proposed architecture implemented on a Xilinx FPGA board, ZCU102. We also compare the proposed architecture with CPU, GPU, and existing FPGA implementations. For the CPU, we use Intel Core i9-9980H, which is a notebook CPU from Intel Corporation, and the computational accuracy is FP32. For the GPU, we use Xilinx's Jetson AGX Xavier GPU for embedded systems, with an accuracy of FP16. The CPU and GPU use Python 3.6 and PyTorch 2.3 libraries for inference, and the batch size is set to 1 to minimize latency. For the existing and proposed FPGA implementations, we used the circuit implemented in the ZCU102 to perform the inference, and the computational accuracy was INT8. In all experiments, we measured the time from the start of the image data transfer to the external memory until the inference results were written to the external memory. We also measured the power of the entire board using an AC watt checker.

Table 5.4 shows the performance and inference speed of each platform. The experiments found that the proposed method is 7.0 times faster than the CPU and 2.3 times faster than the GPU. The inference speed is inferior to the existing FPGA implementation. This is because the current FPGA implementation is a highly parallel circuit that uses more DSP block resources than the proposed method.

5.5 Conclusion

In this chapter, we have proposed a multi-core architecture for fast CNN inference, which has the problem that the achievable operating frequency becomes small when a large HLS design is implemented. The proposed architecture solves this problem by connecting multiple cores designed by HLS as a small design with FIFOs. It is possible to design a circuit with a high operating frequency even when implemented by HLS. Evaluation experiments show that the proposed method is 7.0 times faster than the CPU and 2.3 times faster than the GPU.

Chapter 6

Summary

We have proposed co-design methods for machine learning algorithms and computer architectures to conclude an optimization method for machine learning accelerator on an FPGA. Chapter 2 presented the design and optimization methods of random forests on FPGAs. k-means clustering is used to optimize the algorithm and the design flow based on high-level synthesis design, enabling us to obtain high-performance RF in a short design time. Chapter 3 proposed an efficient indirect memory access architecture using sparse weighted CNNs. We implemented OpenPose, a pose estimation algorithm, on FPGA for the first time, enabling inference of pose estimation that satisfies the timing constraints of real-time applications. Chapter 4 proposed a method for on-chip memory compression in CNN implementation. This research enables us to implement large CNN models on FPGAs with very tiny memory without memory constraints. Chapter 5 proposed a method to realize a large-scale circuit of CNN with high operating frequency by using high-level synthesis. We achieved a 500 MHz implementation despite the large-scale design.

This research has proposed an optimization design flow that satisfies the constraints, including throughput and computational resource limitations, at the slight expense of recognition accuracy for machine learning. It has become possible to develop high-performance implementations quickly by proposing flows using HLS. We have improved the three requirements for autonomous systems: speed, power consumption, and design cost by co-designing the algorithm and hardware and proving FPGA accelerators' high potential. As an implementation method based on our research, we have created a framework allowing a non-expert to realize a high-performance machine learning accelerator on FPGA. The co-design of the machine learning algorithm and architecture and the establishment of the design flow shown in this study will be helpful for future machine learning algorithms or models.

Acknowledgements

I would like to sincerely thank my advisor, Associate Professor Hiroki Nakahara, for his guidance in my research, support in university life, and life advice. Without Associate Professor Hiroki Nakahara, I would not have had a fulfilling graduate school experience. I would like to thank Professor Atsushi Takahashi for supporting my research by giving me a view of my research motivation. I would like to thank Assistant Professor Shimpei Sato for helping and advising my research. I would like to thank the secretary, Mrs. Reiko Shimura, for managing my research expenses.

I am grateful to my colleagues in the laboratory. I would like to thank Mr. Masayuki Shimoda for inspiring me in my research through discussions and daily conversations. I would like to thank Mr. Naoto Soga for providing me with an enjoyable research life by improving the lab atmosphere. I would like to thank Mr. Kouki Sayama for the research discussions we had together. I would like to thank Mr. Ryousuke Kuramochi for his advice on implementing my research. I would like to thank Mr. Takeshi Senoo for helping me run the laboratory's experimental system.

Finally, I would like to express my deepest gratitude to my parents for supporting me all my life. To my mother, I would like to thank you for raising me to be a curious person. To my father, I would like to thank you for accepting my whimsical nature and supporting our family.

References

[1] Xilinx, "Vivado Design Flows Overview." https://japan.xilinx.com/content/xilinx/en/support/documentation-navigation/design-hubs/dh0002-vivado-design-flows-overview-hub.html. [Online; accessed 1-February-2022].

[2] Intel, "Intel FPGA SDK for OpenCL Software Technology." https://www.intel.com/content/www/us/en/software/programmable/sdk-for-opencl/overview.html. [Online; accessed 1-February-2022].

[3] Y. Yang, Q. Huang, B. Wu, T. Zhang, L. Ma, G. Gambardella, M. Blott, L. Lavagno, K. Vissers, J. Wawrzynek, *et al.*, "Synetgy: Algorithm-hardware co-design for convnet accelerators on embedded fpgas," in *International Symposium on Field-Programmable Gate Arrays*, pp. 23–32, 2019.

[4] L. Guo, Y. Chi, J. Wang, J. Lau, W. Qiao, E. Ustun, Z. Zhang, and J. Cong, "Autobridge: Coupling coarse-grained floorplanning and pipelining for high-frequency hls design on multi-die fpgas," in *International Symposium on Field-Programmable Gate Arrays*, pp. 81–92, 2021.

[5] L. Breiman, "Random forests," *Springer, Machine learning*, vol. 45, no. 1, pp. 5–32, 2001.

[6] W. S. McCulloch and W. Pitts, "A logical calculus of the ideas immanent in nervous activity," *Springer The bulletin of mathematical biophysics*, vol. 5, no. 4, pp. 115–133, 1943.

[7] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Nature*, vol. 323, no. 6088, pp. 533–536, 1986.

[8] Y. LeCun, Y. Bengio, *et al.*, "Convolutional networks for images, speech, and time series," *The Handbook of Brain Theory and Neural Networks*, vol. 3361, no. 10, pp. 255–258, 1995.

[9] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Conference on Neural Information Processing Systems*, pp. 1097–1105, 2012.

[10] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[11] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Conference on Computer Vision and Pattern Recognition*, pp. 770–778, 2016.

[12] 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," *arXiv preprint arXiv:1704.04861*, 2017.

[13] J. Hu, L. Shen, and G. Sun, "Squeeze-and-excitation networks," in *Conference on Computer Vision and Pattern Recognition*, pp. 7132–7141, 2018.

[14] M. Tan and Q. V. Le, "Efficientnet: Rethinking model scaling for convolutional neural networks," *arXiv preprint arXiv:1905.11946*, 2019.

[15] M. Ozuysal, M. Calonder, V. Lepetit, and P. Fua, "Fast keypoint recognition using random ferns," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 32, no. 3, pp. 448–461, 2009.

[16] Y. Amit and D. Geman, "Shape quantization and recognition with randomized trees," *MIT Press Neural Computation*, vol. 9, no. 7, pp. 1545–1588, 1997.

[17] D. Tang, Y. Liu, and T.-K. Kim, "Fast pedestrian detection by cascaded random forest with dominant orientation templates.," in *British Machine Vision Conference*, vol. 1, p. 5, Citeseer, 2012.

[18] S. Hinterstoisser, V. Lepetit, S. Ilic, P. Fua, and N. Navab, "Dominant orientation templates for real-time detection of texture-less objects," in *Conference on Computer Vision and Pattern Recognition*, pp. 2257–2264, IEEE, 2010.

[19] J. Shotton, A. Fitzgibbon, M. Cook, T. Sharp, M. Finocchio, R. Moore, A. Kipman, and A. Blake, "Real-time human pose recognition in parts from single depth images," in *Conference on Computer Vision and Pattern Recognition*, pp. 1297–1304, Ieee, 2011.

[20] M. Dantone, J. Gall, G. Fanelli, and L. Van Gool, "Real-time facial feature detection using conditional regression forests," in *Conference on Computer Vision and Pattern Recognition*, pp. 2578–2585, IEEE, 2012.

[21] H. Le, W. Jiang, and V. K. Prasanna, "A sram-based architecture for trie-based ip lookup using fpga," in *International Symposium on Field-Programmable Custom Computing Machines*, pp. 33–42, IEEE, 2008.

[22] T. Becker, Q. Liu, W. Luk, G. Nebehay, and R. Pflugfelder, "Hardware-accelerated object tracking," in *International Conference on Field-Programmable Logic and Applications*, 2011.

[23] B. Van Essen, C. Macaraeg, M. Gokhale, and R. Prenger, "Accelerating a random forest classifier: Multi-core, gp-gpu, or fpga?," in *International Symposium on Field-Programmable Custom Computing Machines*, pp. 232–239, IEEE, 2012.

[24] J. Oberg, K. Eguro, R. Bittner, and A. Forin, "Random decision tree body part recognition using fpgas," in *International Conference on Field-Programmable Logic and Applications*, pp. 330–337, IEEE, 2012.

[25] M. S. Abdelfattah, A. Hagiescu, and D. Singh, "Gzip on a chip: High performance lossless data compression on fpgas using opencl," in *Proceedings of the International Workshop on OpenCL*, pp. 1–9, 2014.

[26] Xilinx, "Xilinx SDSoC Development Environment." `https://www.xilinx.com/products/design-tools/legacy-tools/sdsoc.html`. [Online; accessed 1-February-2022].

[27] R. Narayanan, D. Honbo, G. Memik, A. Choudhary, and J. Zambreno, "An fpga implementation of decision tree classification," in *Design, Automation and Test in Europe Conference and Exhibition*, pp. 1–6, IEEE, 2007.

[28] J. MacQueen *et al.*, "Some methods for classification and analysis of multivariate observations," in *Berkeley Symposium on Mathematical Statistics and Probability*, vol. 1, pp. 281–297, Oakland, CA, USA, 1967.

[29] D. Arthur and S. Vassilvitskii, "k-means++: The advantages of careful seeding," tech. rep., Stanford, 2006.

[30] D. Pelleg, A. W. Moore, *et al.*, "X-means: Extending k-means with efficient estimation of the number of clusters.," in *International Conference on Machine Learning*, vol. 1, pp. 727–734, 2000.

[31] Scikit-leran, "Scikit-leran: Machine Learning in Python." `http://scikit-learn.org/stable/`. [Online; accessed 1-February-2022].

[32] U. Irvine, "UCI Machine Leraning Repository." `https://archive-beta.ics.uci.edu/`. [Online; accessed 1-February-2022].

[33] H. Nakahara, A. Jinguji, T. Fujii, and S. Sato, "An acceleration of a random forest classification using altera sdk for opencl," in *International Conference on Field-Programmable Technology*, pp. 289–292, IEEE, 2016.

[34] Cython, "Cython: C-Extensions for Python." `http://cython.org/`. [Online; accessed 1-February-2022].

[35] W.-T. Lo, Y.-S. Chang, R.-K. Sheu, C.-C. Chiu, and S.-M. Yuan, "Cudt: a cuda based decision tree algorithm," *Hindawi The Scientific World Journal*, vol. 2014, 2014.

[36] S.-E. Wei, V. Ramakrishna, T. Kanade, and Y. Sheikh, "Convolutional pose machines," in *Conference on Computer Vision and Pattern Recognition*, pp. 4724–4732, 2016.

[37] T. Simon, H. Joo, I. Matthews, and Y. Sheikh, "Hand keypoint detection in single images using multiview bootstrapping," in *Conference on Computer Vision and Pattern Recognition*, pp. 1145–1153, 2017.

[38] Z. Cao, T. Simon, S.-E. Wei, and Y. Sheikh, "Realtime multi-person 2d pose estimation using part affinity fields," in *Conference on Computer Vision and Pattern Recognition*, pp. 7291–7299, 2017.

[39] E. Nurvitadhi, J. Sim, D. Sheffield, A. Mishra, S. Krishnan, and D. Marr, "Accelerating recurrent neural networks in analytics servers: Comparison of fpga, cpu, gpu, and asic," in *International Conference on Field-Programmable Logic and Applications*, pp. 1–4, IEEE, 2016.

[40] E. Nurvitadhi, D. Sheffield, J. Sim, A. Mishra, G. Venkatesh, and D. Marr, "Accelerating binarized neural networks: Comparison of fpga, cpu, gpu, and asic," in *International Conference on Field-Programmable Technology*, pp. 77–84, IEEE, 2016.

[41] E. Nurvitadhi, G. Venkatesh, J. Sim, D. Marr, R. Huang, J. Ong Gee Hock, Y. T. Liew, K. Srivatsan, D. Moss, S. Subhaschandra, *et al.*, "Can fpgas beat gpus in accelerating next-generation deep neural networks?," in *International Symposium on Field-Programmable Gate Arrays*, pp. 5–14, 2017.

[42] Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3, pp. 367–379, 2016.

[43] H. Yonekawa, S. Sato, and H. Nakahara, "A ternary weight binary input convolutional neural network: Realization on the embedded processor," in *International Symposium on Multiple-Valued Logic*, pp. 174–179, IEEE, 2018.

[44] D. Nguyen, D. Kim, and J. Lee, "Double mac: Doubling the performance of convolutional neural networks on modern fpgas," in *Design, Automation and Test in Europe Conference and Exhibition*, pp. 890–893, IEEE, 2017.

[45] T. Jin and S. Hong, "Split-cnn: Splitting window-based operations in convolutional neural networks for memory system optimization," in *International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 835–847, 2019.

[46] Google, "Edge TPU performance benchmarks." `https://coral.ai/docs/edgetpu/benchmarks/`. [Online; accessed 8-January-2021].

[47] K. Guo, L. Sui, J. Qiu, S. Yao, S. Han, Y. Wang, and H. Yang, "From model to fpga: Software-hardware co-design for efficient neural network acceleration," in *Symposium on High Performance Chips*, pp. 1–27, IEEE, 2016.

[48] A. Podili, C. Zhang, and V. Prasanna, "Fast and efficient implementation of convolutional neural networks on fpga," in *International Conference on Application-specific Systems, Architectures and Processors*, pp. 11–18, IEEE, 2017.

[49] S. Yin, S. Tang, X. Lin, P. Ouyang, F. Tu, L. Liu, and S. Wei, "A high throughput acceleration for hybrid neural networks with efficient resource management on fpga," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 38, no. 4, pp. 678–691, 2018.

[50] M. Zhu and S. Gupta, "To prune, or not to prune: exploring the efficacy of pruning for model compression," *arXiv preprint arXiv:1710.01878*, 2017.

[51] E. Elsen, M. Dukhan, T. Gale, and K. Simonyan, "Fast sparse convnets," in *Conference on Computer Vision and Pattern Recognition*, pp. 14629–14638, 2020.

[52] Xilinx, "Vitis Unified Software Platform." `https://www.xilinx.com/products/design-tools/vitis/vitis-platform.html`. [Online; accessed 1-February-2022].

[53] 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," in *International Conference on Field-Programmable Logic and Applications*, pp. 136–143, IEEE, 2019.

Publications

Jpurnal Papers

1. A. Jinguji, S. Sato, and H. Nakahara, " An FPGA Realization of a Random Forest with k-means Clustering using a High-level Synthesis Design, " IEICE Transactions on Information and Systems, Vol. E101-D, No. 2, pp. 354-362, February, 2018.

2. A. Jinguji, S. Sato, and H. Nakahara, "Weight Sparseness for a Feature-Map-Split-CNN Toward Low-Cost Embedded FPGAs,"IEICE Transactions on Information and Systems, Vol. E104-D, No. 12, pp. 2040-2047 December, 2021.

International Conferences

1. H. Nakahara, A. Jinguji, T. Fujii, and S. Sato, " An Acceleration of a Random Forest Classification using Altera SDK for OpenCL, " International Conference on Field-Programmable Technology, pp. 285-288, Xian, China, December, 2016.

2. H. Nakahara, A. Jinguji, S. Sato and T. Sasao, " A Random Forest using a Multi-valued Decision Diagram on an FPGA, " International Symposium on Multiple-Valued Logic, pp. 266-271, Novi Sad, Serbia, May, 2017.

3. A. Jinguji, T. Fujii, S. Sato and H. Nakahara, " An FPGA Realization of OpenPose based on a Sparse Weight Convolutional Neural Network," International Conference on Field-Programmable Technology, pp. 313-316, Naha, Okinawa, Japan, December, 2018.

4. H. Nakahara, A. Jinguji, M. Shimoda and S. Sato, " An FPGA-based Fine-Tuning Accelerator for a Sparse CNN, " International Symposium

on Field-Programmable Gate Arrays, pp. 186-186, Seaside, California, USA, February, 2019.

5. H. Nakahara, Y. Sada, M. Shimoda, K. Sayama, A. Jinguji, S. Sato, “FPGA-based Training Accelerator Utilizing Sparseness of Convolutional Neural Network,”International Conference on Field-Programmable Logic and Applications, pp. 180-186, Barcelona, Spain, September, 2019.

6. A. Jinguji, Y. Sada, H. Nakahara, “Real-Time Multi-Pedestrian Detection in Surveillance Camera using FPGA,”International Conference on Field-Programmable Logic and Applications, pp. 424-425, Barcelona, Spain, September, 2019.

7. Y. Sada, M. Shimoda, A. Jinguji, H. Nakahara, “A Dataflow Pipelining Architecture for Tile Segmentation with a Sparse MobileNet on an FPGA,”International Conference on Field-Programmable Technology, pp. 267-270, Tianjin, China, December, 2019.

8. H. Nakahara, Q. Zhiqiang, A. Jinguji, W. Luk, “R2CNN: Recurrent Residual Convolutional Neural Network on FPGA,”International Symposium on Field-Programmable Gate Arrays, pp. 319-319, Seaside, California, USA, February, 2020.

9. A. Jinguji, S. Sato, H. Nakahara, “Tiny On-Chip Memory Realization of Weight Sparseness Split-CNNs on Low-end FPGAs,”International Symposium on Field-Programmable Custom Computing Machines, pp. 229-229, Online, May, 2020.

10. Y. Sada, N. Soga, M. Shimoda, A. Jinguji, S. Sato and H. Nakahara, “Fast Monocular Depth Estimation on an FPGA,”International Parallel and Distributed Processing Symposium Workshops, pp. 143-146, Online, May, 2020.

11. T. Senoo, A. Jinguji, R. Kuramochi and H. Nakahara, “A Multilayer Perceptron Training Accelerator using Systolic Array,”Asia Pacific Conference on Circuits and Systems, No. 1570752115, Online, November, 2021.

Japanese Domestic Conferences

1. 中原啓貴, 神宮司明良, 藤井智也, 佐藤真平, 丸山直也, “Altera SDK for OpenCL を用いたランダムフォレストによる分類の高速化,”コンピュー

タシステム研究会, CPSY2016-25, Vol.116, No.177, pp. 175-180, 長野, 2016 年 8 月.

2. 中原啓貴, 神宮司明良, 藤井智也, 佐藤真平, 丸山直也, “Altera SDK for OpenCL を用いた組込みメモリに基づくランダムフォレストによる分類について,”リコンフィギャラブルシステム研究会, Vol. 116, No. 210, pp. 57-62, 富山, 2016 年 9 月.

3. 中原啓貴, 神宮司明良, 佐藤真平, 笹尾勤, 丸山直也, “ 多値決定グラフを用いたランダムフォレストに関して, ”第 39 回多値論理フォーラム, 多値論理研究ノート第 39 巻, 岩手, 2016 年 9 月.

4. 神宮司明良, 佐藤真平, 中原啓貴, “ 特徴空間の分割に k 平均法を導入したランダムフォレストの FPGA 実装, ”第 30 回多値論理とその応用研究会, 金沢, 2017 年 1 月.

5. 神宮司明良, 佐藤真平, 中原啓貴, “Feature-Map Separable Convolution による小メモリ FPGA での画像認識の実現, ”リコンフィギャラブルシステム研究会, RECONF2018-41, pp. 39-44, 広島, 2018 年 12 月.

6. 神宮司明良, 下田将之, 中原啓貴, “ 特徴マップを空間分割した CNN の FPGA における小メモリ実装,”VLSI 設計技術研究会, VLD2018-94, pp. 7-12, 沖縄, 2019 年 3 月.

7. 神宮司明良, 下田将之, 中原啓貴, “ 特徴マップを空間分割した CNN の FPGA における小メモリ実装について, ”リコンフィギャラブルシステム研究会, RECONF2019-16, pp. 85-90, 東京, 2019 年 5 月.

8. 神宮司明良, 佐藤真平, 中原啓貴, “ Wide-SIMD を用いた ISA ベースのスパース CNN の FPGA 実装, ”リコンフィギャラブルシステム研究会, RECONF2019-37, pp. 9-14, 愛媛, 2019 年 11 月.

9. 佐山功起, 神宮司明良, 曽我尚人, 中原啓貴, “ 解像度に基づくスケールが可能な CNN アクセラレータの FPGA 実現に関して, ”リコンフィギャラブルシステム研究会, RECONF2020-68, pp. 58-62, オンライン, 2021 年 1 月.

10. 神宮司明良, 中原啓貴, “ 高位合成を用いたマルチコア構成のニューラルネットワークの FPGA 実装, ”リコンフィギャラブルシステム研究会, RECONF2021-18, pp. 7-12, オンライン, 2021 年 9 月.

11. 神宮司明良, 中原啓貴, “ gMLP を用いた画像認識向け DNN アクセラレータの FPGA 実装,”リコンフィギャラブルシステム研究会, RECONF 2021-29, pp. 25-30, オンライン, 2021 年 12 月.

12. 妹尾豪士, 神宮司明良, 倉持亮佑, 中原啓貴, “シストリックアレイによる多層パーセプトロンの学習アクセラレータについて,”リコンフィギャラブルシステム研究会, RECONF2021-31, pp. 37-42, オンライン, 2021年12月.

13. 市川雄樹, 神宮司明良, 倉持亮佑, 中原啓貴, “蒸留とレイヤー枝刈りによるエッジデバイス推論処理の高速化について,”リコンフィギャラブルシステム研究会, RECONF2022-66, pp. 49-54, オンライン, 2022年1月.

14. 高嶋優希, 神宮司明良, 中原啓貴, “最終層学習によるDPUの学習機能追加について,”リコンフィギャラブルシステム研究会, RECONF2021-67, pp. 55-60, オンライン, 2022年1月.

Awards

1. リコンフィギャラブルシステム研究会 若手優秀講演賞, “Feature-Map Separable Convolutionによる小メモリFPGAでの画像認識の実現,”電子情報通信学会リコンフィギャラブルシステム研究会, 2019年1月.

2. デザインガイア 優秀ポスター発表賞, “gMLPを用いた画像認識向けDNNアクセラレータのFPGA実装,”電子情報通信学会デザインガイアポスター賞選奨実行委員会, 2021年12月.

3. リコンフィギャラブルシステム研究会 若手優秀講演賞, “高位合成を用いたマルチコア構成のニューラルネットワークのFPGA実装,”電子情報通信学会リコンフィギャラブルシステム研究会, 2022年1月.