Master's Thesis

Optimized-FDanQ: Implementation of Hybrid Neural Network "DanQ" on Cloud Multi-FPGA and its Optimization under Given Costs

Takahiro Inaba

Supervisor: Prof. Yasushi Inoguchi

School of Information Science
Japan Advanced Institute of Science and Technology
(Master of Science)

March, 2021

Abstract

In this information society, the amount of data is rapidly increasing, especially in the field of Astronomy, Twitter, Youtube, and Genomics. In these four fields, storage for genomics is increasing the most, and the way to process it fast has been one of the big tasks for a long time. There are so many DNA sequences that people have to work on. One of the genome analysis that people have to face is to find the function of DNA from a DNA sequence. Recently, Machine Learning has been used to find the function of DNA from a DNA sequence. However, training a machine learning model for DNA sequences takes much time due to the size of the dataset. Besides, since DNA sequences are represented by four types of the base, which are Adenine, Guanine, Thymine, and Cytosine, it can be represented by the bit-width of two. FPGA has a substantial advantage of processing these kinds of string operations because FPGA can construct a dedicated state machine. Also, FPGA can be a useful resource for processing fast by pipelining.

In addition, more and more companies are using cloud services such as AWS for their acceleration. Since cloud users always have to consider the trade-off between execution time and cloud instance usage fee, it is necessary to optimize these two things depending on each cloud user.

In this paper, we propose the following two ideas.

- Mutli-FPGA Implementation
- Cloud Optimization under Give Costs

We tried to accelerate a deep learning model called DanQ using FPGAs. It is said that FPGA is sufficient for data such as genomics data because DNA sequence can be represented by 1 bit and does not require a large bit-width for processing. We mainly focused on a BiLSTM layer, which is the most time-consuming part of the DanQ model. We quantized the parameters of the BiLSTM layer to the bit width of 16 in order to implement on FPGA without losing the training accuracy. We also implemented the BiLSTM layer to multiple FPGAs to obtain a better execution time. As a result, we could accelerate the DanQ model by using a single FPGA by 1.05x compared to our CPU implementation. Besides, our implementation on 8 FPGAs gets 2.87x faster than the dual FPGA implementation and 6.00x faster than the CPU implementation.

Also, our implementations can change the resource size during the execution to optimize the execution time or cloud instance usage fee depending on the users' needs. Comparing a case of using 8 FPGAs for all time and a case in which we optimized the number of FPGAs during the training with our model, we obtained the result that we can save the cloud usage fee for 56.28% by only taking 16.00% extra time.

Contents

List of Figures

List of Tables

List of abbreviations

FNN Feedforward Neural Network

CNN Convolutional Neural Network

RNN Recurrent Neural Network

FPGA Field Programmable Gate Array

AWS Amazon Web Service

LSTM Long Short-Term Memory

GRU Gated Recurrent Units

BiLSTM Bidirectional Long Short-Term Memory

ReLU Rectified Linear Unit

RAM Random Access Memory

BRAM Block Random Access Memory

URAM Ultra Random Access Memory

BLE Basic Logic Element

FF Flip Flop

SRAM Static Random Access

DSP Digital Signal Processing Memory

HLS High Level Synthesis

SGD Stochastic Gradient Descent

SLR Super Logic Region

RMS Root Mean Square

IDE Integrated Development Environment

RTL Register Transfer Level

AWS Amazon Web Service

HDL Hardware Description Language

DMA Direct Memory Access

Chapter 1

Introduction

1.1 Research Background

In this Information era, the amount of data is increasing rapidly in the society. Especially, The amount of information in the field of astronomy, twitter, youtube, genomics is massive. It is getting easier and easier for people to upload anything they have in the internet and that leads to a huge increase of data in the society. Among these field, the amount of increasing data from genomics which increases with 2-40(EB) per year are the highest followed by Youtube which increases 1-2(EB) per year [1]. Recently, deep learning has proved its potential in graphic processing such as segmentation and motion estimation, and speech recognition. Moreover, it is also used in genomics and showed results with high accuracy. However, as we mentioned above, the amount of genome data is massive and that makes the training time very long.

Moreover, there are so many situation for users who use cloud services for acceleration. They sometimes want to finish the training fast no matter how much it cost, while some other time they might want to save the usage fee even if the training time is very long. Therefore, it is a task to optimize the training time depending on the users needs by changing the size of resources that are used for the acceleration during the training.

1.2 Research Purpose

The purpose of this research is to accelerate a training of a deep learning for genome data by implementing on multi-FPGA. The main focus of our research is an optimization using AWS EC2 F1 instance in AWS cloud service which has multiple FPGA. By changing the size of instances, we can optimize the training time as well as instance usage fee depending on a user's needs.

1.3 Audience

The audience of this research is people who are familiar with DNA sequence such as researchers and investigators. In addition, people who are interested in deep learning or hardware implementation would be the audience. This paper will state Convolutional Neural Network (CNN) as well as Recurrent Neural Network (RNN), and their implementations on one or more hardwares.

1.4 Contribution

There are so many advantages using FPGA in terms of flexibility and power efficiency. Therefore, we can estimate that training time of the deep learning model will be faster by implementing on FPGA. In addition, in case of using instances from cloud service, we can optimize the training time and instance usage fee depending on a user's needs by changing the instance size during the training. Hence, this research makes a contribution to users who use cloud services as acceleration.

1.5 Motivation

A process using bigdata leads to huge processing time and many researchers have been working on it. Especially, it is hard to accelerate the process when it is a complex application. However, there is a chance to accelerate the training time by doing distributed training using multiple devices such as FPGAs. Moreover, some companies are relying on cloud services for their acceleration and there is always a discussion of the trade-off between the size of the resources and the usage fee. Thus, it is worth optimizing the training time and the instance usage fee for each users.

1.6 Paper Goals

Training a deep neural network (DNN) for DNA sequence takes a lot of time due to the size of the dataset. Thus, it is one of the big task to process the massive data in genomics efficiently. In our approach, we accelerated the training by using multiple FPGAs. Also, we provide cloud users an optimized training method by changing the size of the instance during the training depending on the instance fee which fluctuates in the daytime and the nighttime. Therefore, Our goals of this paper are accelerating the training using multiple FPGAs and the optimization of training time and the cloud instance usage fee by changing the size of the instance during the training.

1.7 Organization

This paper is mainly organized as 7 chapters. Following chapters are Background, Related Works, Multi-FPGA Implementation, Cloud Optimization, Conclusion and Future Works, and Acknowledgment. In the chapter of Background, we explain the

basic idea of neural network such as CNN and RNN. The NN model using genome data as well as the related research are explained in the next chapter. Our two proposed ideas of this paper are mentioned in Chapter 4 and 5, respectively. In each two chapters, we indicate the proposed idea and the evaluation. We conclude this paper and state our future plans in Chapter 6.

Chapter 2

Background

2.1 Introduction

In this chapter, we explain basic ideas of neural network (NN) which is used mainly for this research. There are so many types and techniques in NN. The explanation below shows one of them.

2.2 Neural Network

2.2.1 Feedforward Neural Network

Feedforward neural network (FNN) is one of simplest and most used neural networks (NN), which is consisted of many compute units in each layer. It has forward process and backward process and a NN which has many layers is called deep neural network (DNN). In the forward process, activation, which is a input of a layer, is multiply-accumulated with weights. Let x be the input to a layer, and let w be its weight. An example of a output of the layer y with the bias b can be shown below:

$$y = w_1x_1 + w_2x_2 + w_3x_3 + b. \tag{2.1}$$

This equation is an example of three inputs to a unit. After this process, activation function is applied and final output of each unit is shown in the equation, where $f()$ represent a activation function.

$$z = f(y) \tag{2.2}$$

The main reason for applying an activation function is to make it easier to train and to gain high accuracy. Some activation functions make the outputs from each layers as nonlinear value to improve the expressiveness, while some other activation functions prevent outputs of a layer becoming too large values and try to make them 0, which leads to efficient training.

There are many variety of functions that are used for activation function. For instance, here is a activation function called Sigmoid function as shown in Equation

2.3, which has been used for long time. However, there is a big problem that value of gradient become zero in the back propagation phase, specially when the NN has so many layers. This problem is known as Vanishing Gradient Problem. It is more likely to happen when we choose the Sigmoid function as an activation function. This is because the output of the derivative of Sigmoid function has 0.25 as maximum value and the value of gradient is going to be smaller and smaller if we pile up these kind of layers.

$$y = 1/(1 + exp(-x)) \tag{2.3}$$

Nowadays, the most popular activation function is called Rectified Linear Unit (ReLU) as shown in Equation 2.4, which outputs 0 for the negative inputs and outputs the inputs for the positive inputs. This is introduced not only for solving the Vanishing Gradient Problem but also to have better processing time by making the activation sparse to compute more simply.

$$y = \begin{cases} x & (x \geq 0) \\ 0 & (x < 0) \end{cases} \tag{2.4}$$

Figure 2.1 shows the comparison between Sigmoid function and ReLU function. It is easier to understand that the outputs for Sigmoid function have 0 for minimum value and 1 for maximum value. On the other hand, ReLU function has so many 0 values for the outputs and has no maximum value, while the minimum value and the maximum value for its derivative are 0 and 1, respectively.

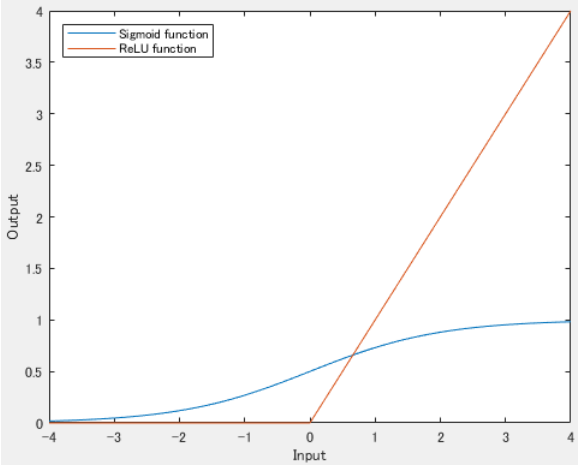

Figure 2.1: Examples of activation functions

NN consists of multiple layers which have multiple units. The training is processed by optimizing these weights to have smaller loss. The loss is generated by a loss function such as binary cross-entropy:

$$loss = -t[i]log(out[i]) - (1 - t[i])log(1 - out[i]), \tag{2.5}$$

where t represents the correct output and out represents the real output, with each elements of i.

In the backward process, which is also called backward propagation, there are two parameters called gradient and error. From the last layer, derivatives of each function in each layer are computed by the chain rule of calculus to the first layer. The parameter that goes from last layer to the first layer is called an error. At the same time, the back propagation compute the parameter called an gradient which is used for updating the weight in each layers.

There are many algorithms for updating the weights which we call them an optimizer. One of the famous optimizer is Stochastic Gradient Descent (SGD). SGD is used to perform learning using the gradient which is obtained in backward propagation phase. Therefore, the weights are decreased or increased depending on the gradient and the degree of it is determined by learning rate which we decide in advance. The equation is shown in Equation 2.6.

$$W_{t+1} = W_t - \alpha(\delta L(W_t))/(\delta W_t) \tag{2.6}$$

Where suffix of weight represents the time and the α represents the learning rate. If the learning rate is too big, it is difficult to get the best weight values. On the other hand, if the learning rate is too small, it takes so mush time to get the best weight values. $(\delta L(W))/(\delta W)$ is differentiate of loss function by the weight, which represents the gradient.

To change the learning rate during the training, new optimizer called Adagrad was introduced by John Duchi, Elad Hazan, and Yoram Singer [6]. This optimizer has a new parameter h to change the learning rate as shown in Equation 2.7. This parameter h keeps and remembers the last gradient in order to know the appropriate learning rate. The update equation is shown in Equation 2.8.

$$h_{t+1} = h_t + (\delta L(W_t))/(\delta W_t) \tag{2.7}$$

$$W_{t+1} = W_t - \alpha(1/(\sqrt{h_{t+1}}))(\delta L(W_t))/(\delta W_t) \tag{2.8}$$

The biggest advantage of using Adagrad is that the learning rate changes depending on the progress of training. However, learning rate can only be decreased and cannot be increased.

There is an improved optimizer for Adagrad called RMSprop which is introduced by Geoffrey Hinton, Nitish Srivastava, and Kevin Swersky [9]. This is introduced to overcome the problem in Adagrad that learning rate decreases rapidly. Equation 2.10 shows the update equation for RMSprop.

$$E_t = \gamma E_{t-1} + (1-\gamma)(\delta L(W_t))/(\delta W_t)^2 \tag{2.9}$$

$$W_{t+1} = W_t - \alpha(1/(\sqrt{E_t + \epsilon}))(\delta L(W_t))/(\delta W_t) \tag{2.10}$$

2.2.2 Convolutional Neural Network

The Convolutional Neural Network (CNN) is a FNN which consists of one or more sets of Convolutional layer and Pooling layer. In general, CNN also has Full-connected layers which we mentioned above. CNN is trained by optimizing the weight using back propagation like FNN and is often used for image recognition. The main difference between CNN and FNN is that CNN has specific connections between units in adjacent layers, while FNN has all connections between units in adjacent layers. The processes of convolution and pooling is the main part of CNN. In convolutional process, it can extract features from the inputs such as images by using the locality of the inputs. For instance, a pixel in a image has relativity with adjacent pixels to a certain extent. CNN can extract feature using these locality. Let y is the output of convolutional layer and z is the input to the layer, which we call it feature map. The equation of this process is shown below:

$$y_{ijm} = \sum_{k=0}^{K-1} \sum_{p=0}^{H-1} \sum_{q=0}^{H-1} z_{i+p,j+q,k}^{(l-1)} h_{pqkm} + b_{ijm} \tag{2.11}$$

Here, h is a filter sized $H \times H$, which is also called weights, and b is a bias. K represents the number of channels of $W \times W$ image and the filter. z and y are inputs and outputs, respectively, with a layer number l. The suffixes i, j, m and p, q, k represent the positions of a pixel of output and each filter, respectively. Figure 2.2 shows an overview of the convolution process.

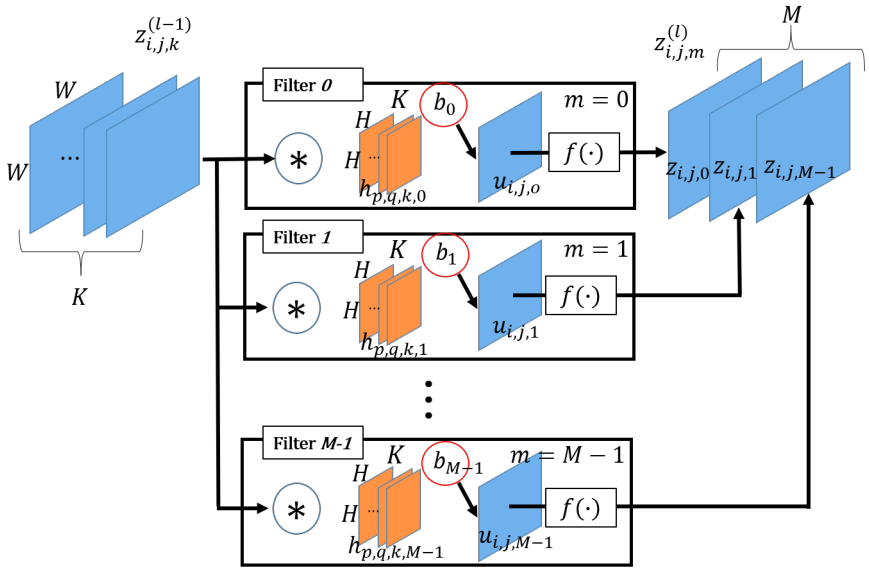

Figure 2.2: The Overview of a Convolutional Layer

When the inputs for the CNN are time series, a process of one dimension convolutions is usually applied which we call it 1-D CNN. What we mentioned above is for convolutional process for two dimensions. In 1-D CNN, which is also used in our research, has a one dimension of filter instead of two dimensions. The inputs can be considered as an array and the filter slides horizontally over the inputs to get outputs, just like a version of two dimensions.

Pooling layer is often used right after the convolutional layer. In this process, this layer outputs the average or maximum value from the inputs window. The window size is determined by each pooling layer. This layer is used to get robustness for the change of the position of each input value. Since the size of the feature map gets smaller, the amount of calculation also decreases by using the Pooling layer.

2.2.3 Recurrent Neural Network

Recurrent Neural Network (RNN) is a NN which is specialized to the time series input data. It remembers the past inputs and outputs data using these past inputs. Simple RNN has a problem that requires huge memory to remember all the past inputs. Therefore, there is a improved RNN called Long Short-Term Memory (LSTM) [11] which decides to remember the inputs or not in order to solve the Vanishing Gradient Problem and Exploding Gradient Problem [2]. In order to do that, LSTM has three different gates: Output gate, Forget gate and Input gate. It also has a cell called memory cell which memorize inputs. The overview of LSTM cell is shown in Figure 2.3

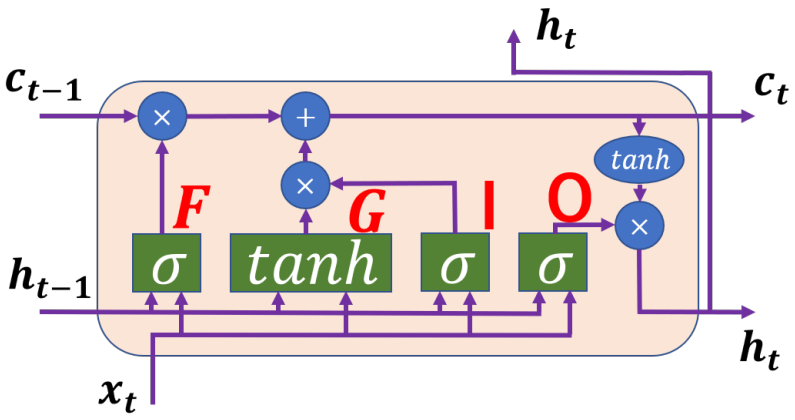

Figure 2.3: The Overview of a LSTM Cell

Here, σ represents the Sigmoid function and, F, G, I, O represents gates and a cell. Also, x, h, c in the Figure 2.3 represent inputs, outputs from the past and outputs of memory cell, respectively. Their suffixes indicate the time of time series input data. The equation of each gates and a memory cell is shown below, where c_t represents outputs of memory cell and h_t represents a hidden state at the time of t.

$$f_t = \sigma(W_f\,[h_{t-1}, x_t] + b_f) \tag{2.12}$$

$$i_t = \sigma(W_i[h_{t-1}, x_t] + b_i) \tag{2.13}$$

$$g_t = tanh(W_g[h_{t-1}, x_t] + b_g) \tag{2.14}$$

$$c_t = \sigma(f_t c_{t-1} + i_t * g_t) \tag{2.15}$$

$$o_t = \sigma(W_o[h_{t-1}, x_t] + b_o) \tag{2.16}$$

$$h_t = o_t * tanh(c_t) \tag{2.17}$$

Figure 2.4 shows the overview of a LSTM layer. Each LSTM block represent a process that we mentioned above. Outputs of an LSTM block is inputs for the next LSTM block and this is an architecture, which has a better performance for time series input data, that is completely different with an architecture of Convolutional layer.

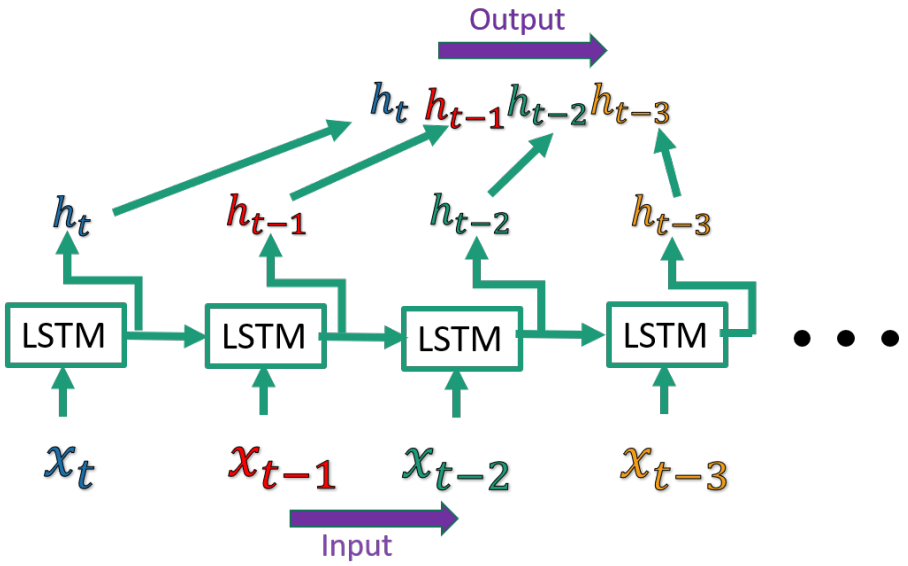

Figure 2.4: The Overview of a LSTM layer

There is a layer called Bidirectional Long Short-Term Memory (BiLSTM) which has two LSTMs which reads the inputs in different way. One LSTM reads the input from past to present which is a normal LSTM, while the other LSTM reads the input from present to past. Since the outputs of these two LSTM are concatenated, the size of outputs of BiLSTM layer is as double as that of the normal unidirectional LSTM. Figure 2.5 shows an overview of the BiLSTM layer with an example of inputs whose size is four (x_t, x_{t+1}, x_{t+2}, x_{t+3}).

In this example, since the size of inputs is four, the size of the outputs is eight. The biggest advantage of using the BiLSTM other than normal unidirectional LSTM is that the training accuracy increases by not only reading from the past to the present but also the present to the past. A normal unidirectional LSTM can only remember and take account of past inputs. However, on the other hand, BiLSTM layer can remember and take account of past and future inputs. Thus, taking future inputs into consideration leads to higher training accuracy comparing to an unidirectional LSTM.

There is also an improved LSTM called a Gated Recurrent Units (GRU). It has less gates than an LSTM which leads to fewer calculation and smaller memory utilization. There are two gates called Update gate and Reset gate. The Update gate acts similar to forget and input gate of an LSTM which make a decision of whether an input should be remembered or not. The Reset gate is a gate that is used to decide how far of the input should be forgotten.

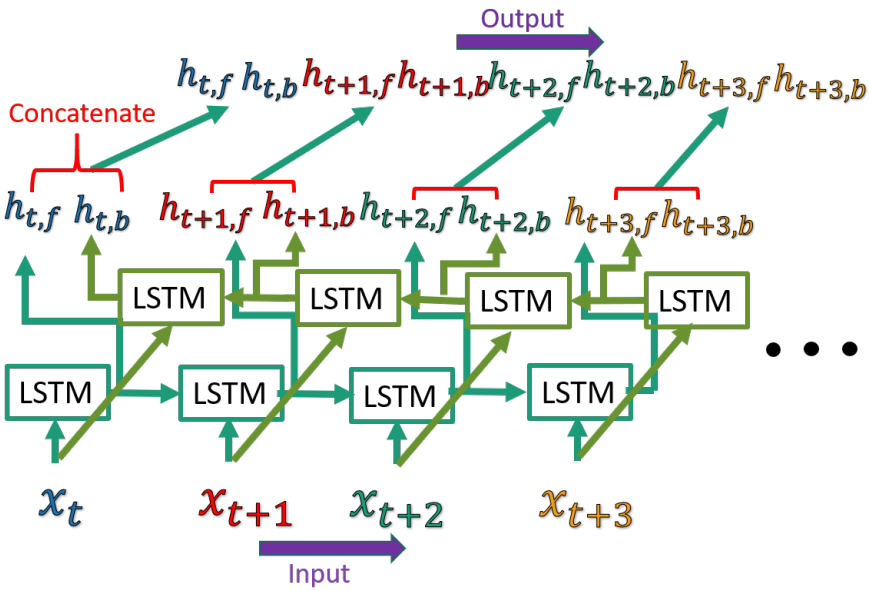

Figure 2.5: The Overview of Bidirectional LSTM layer

2.2.4 Techniques for Training the Model

There are much research that improve the training for deep learning. One of the techniques for prevent overfitting is regularization such as Weight Decay. Overfitting is likely to happen when the value of weights are too large. To avoid the weights to have large value, squared L^2 norm of the weights is added to the loss function. The equation is shown in Equation 2.18, where J and L represents a criterion and loss function with weight w, respectively.

$$J(w) = L(w) + \lambda w^T w \tag{2.18}$$

Here, λ indicates a hyper parameter that controls strength of regularization. By having this idea of adding squared L^2 norm to the loss function, a criterion J is going to be increased when the weights become larger. Therefore, it is possible not to have too large values of weights and this regularization leads to prevent the overfitting. Another known techniques for overfitting is a technique called Dropout, which is introduced by Srivastava et al. in 2014 [10]. This is a technique that some units are removed randomly from the network for only the training. This means no more activations for removed units propagate to next units. Using this technique, the training does not strongly

depends on the training data and this leads to a high generalization performance. In the inference phase, units are not removed and all the units propagate their activations. However, the activations are multiplied by the dropout ratio, which is a ratio of removed units in the training phase. Figure 2.6 shows an example of standard hidden layers which has four units in each layers and Figure 2.7 shows the hidden layers after applying the Dropout.

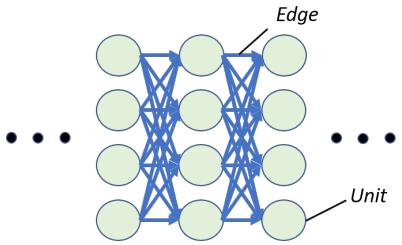

Figure 2.6: A Standard Neural Network

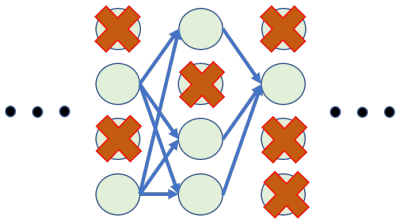

Figure 2.7: Neural Network after applying Dropout

Weight Decay does not work well when a model become very complicated. However, the dropout works well even when a model is huge and complicated.

2.3 Field Programmable Gate Array

Field Programmable Gate Array (FPGA) is one of the Programmable Logic Device (PLD) to freely realize a logic circuit. It consists of three parts: logic block, I/O block, and switch block. Basic Logic Element (BLE) is a fundamental element in a logic block which consists of Look-Up Table (LUT) and Flip Flop (FF). LUT can realize a

combination circuit by keeping a truth table of a desired function in less memory. FF is a memory cell of 1 bit used as a memory cell of a sequential circuit. Figure 2.8 shows an overview of the most popular configuration of an FPGA.

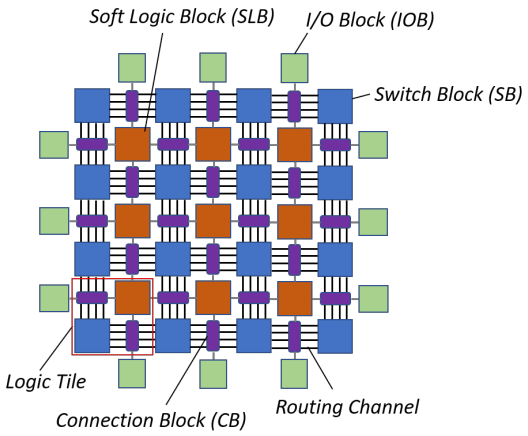

Figure 2.8: The overview of FPGA

Since it is difficult and time-consuming for wiring in terms of multiplication using LUT, a hardware block called Digital Signal Processing (DSP) block has been introduced which is specialized in complex process such as multiplication. Some of the FPGAs such as Xilinx Alveo U200 acceleration cards have regions called Super Logic Region (SLR). An SLR is an FPGA die slice and includes: LUTs, Registers, I/O Components, Gigabit Transceivers, Block Memory, and DSP Blocks. In addition, not only the BRAM but also the URAM can be used for implementation to Xilinx FPGA. Basically, the URAM has more capacity than the BRAM, and an FPGA that has the URAM such as Xilinx UltraScale+ device has an advantage that it has a large capacity in on-chip memory compared to the other FPGA which does not have the URAM, which leads to being able to implement larger size of an application easily.

2.4 High Level Synthesis

High Level Synthesis (HLS) is to generating Hardware Description Language (HDL) from high-level language such as C/C++ and java by using a designated compiler. Thus, it is unnecessary to code using HDL which requires high knowledge of hardware structure and design. Moreover, it is said that productivity of using HDL and high-level language is so much different. It is is much easier for developers to code using high-

level language in terms of that they do not have to describe it precisely because HLS compiler does that instead of the developers. In addition, when we develop using HDL, the HDL design is strongly depends on the hardware that we implement which leads to poor portability while HLS design has rich portability thanks to the HLS compiler.

As we mentioned in Chapter 3, the paper [15] showed a result that there are no so much difference in execution time between a design using HLS tools and a design using HDL. Therefore, we decided to use HLS tools to increase the productivity for the implementation and focus on the optimization.

2.5 Summary

In this Chapter, we introduced the basic mechanism of NN, including CNN and RNN. CNN and RNN are the most important parts of our target model DanQ. Also, we explained the basic knowledge about FPGA as well as the HLS which is used for an FPGA implementation. These knowledge are needed for not only my research but also for related works that are explained in the next chapter.

Chapter 3

Related Works

3.1 Introduction

Since there are huge data in genomics as we mentioned in Chapter 1, DNA sequences that people have to analyze are numerous. Thus, many researchers have been working on it for long time. One of the genomic analysis that people are working on is identifying functional effect of non-coding variants from a DNA sequence. A problem that people have to face is how people can process the huge DNA sequence efficiently. We introduce two research that tried to solve the problem using deep learning.

In addition, there are some papers about an implementation of deep learning on FPGAs. An FPGA can perform greater than a GPU or a CPU in terms of their execution time and power efficiency depending on how they implemented. We introduce the paper in detail.

Moreover, there are increasing number of researchers and companies who are using cloud service to process their application. Using cloud service has big advantages that their resources have scalability and flexibility. Also, developers do not have to care about the maintenance as well as the security of the system. There are some researches about a deep learning using this advantage of cloud service. One of them is mentioned below.

Also, many cloud users always have to consider about the trade-off between the public cloud usage fee and the completion time. In this Chapter, we also introduce about the scheduling methods in order to gain the optimized cloud usage fee as well as the optimized completion time for each cloud users.

3.2 DeepSEA: CNN model for genomics data

In order to find the chromatin effects from DNA sequence, CNN model called DeepSEA was proposed by Jian Zhou and Olga G Troyanskaya [17]. The overview of DeepSEA is shown in figure 3.1.

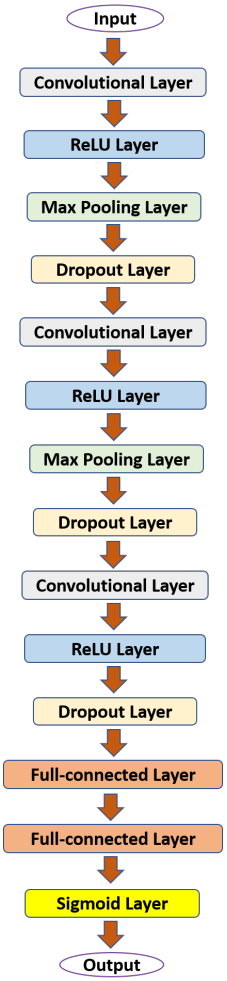

Figure 3.1: The Overview of a DeepSEA model

The size of the inputs to this model is $4 * 1000$ which represents the length of 1000 of DNA sequence and its output size is 919 which is the binary data representing the 919 predicted chromatin features. Input is binary data and is represented by one hot vector, with the columns corresponding to the 4 different base, which are adenine (A), guanine (G), thymin(T) and cytosine (C). Main parts of this model are 3 1d-convolutional layers which extract the pattern from the DNA sequence. After each convolutional layer, it has ReLU which all negative activations is output as zero. It also has two max pooling layer which outputs maximum value in each windows. Three dropout layers regularize the data to prevent over-training and increase the training efficiency. At Last, two full-connected layer summarize the data and sigmoid layer outputs the prediction.

3.3 DanQ: Hybrid model for genoimcs data

Improved model called DanQ is proposed y Daniel Quang and Xiaohui Xie [14].The model has improved in accuracy compare to the DeepSEA. The overview of DanQ is shown in figure 3.2.

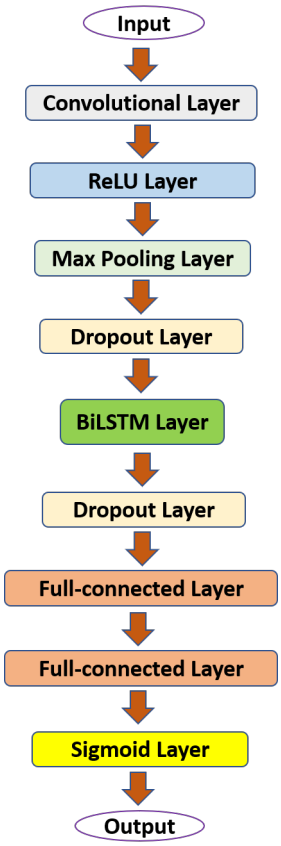

Figure 3.2: The Overview of DanQ model

The biggest difference between DeepSEA and DanQ is whether it has RNN or not. DanQ is a hybrid neural network including both CNN and RNN.It has one convolutional layer and one RNN layer instead of having 3 convolutional layers like DeepSEA. For RNN layer, Bidirectional LSTM layer is used which has two LSTMs which read the inputs from past to present and from present to past. By adding RNN layer to the model, the model can not only capture the feature of the DNA sequence by convolutional layer but also remember the past input by RNN which leads to higher accuracy. The loss and accuracy of the DanQ is shown in Table 3.1 with that of DeepSEA. Theses results are obtained from the training for 60 epochs with the dataset of 4,400,000, where one input is a DNA sequence which has a length of 1000.

Table 3.1: Loss and Accuracy for models

model	Loss	Accuracy
DeepSEA	0.0551	98.20%
DanQ	0.0539	98.23%

3.4 RNN Implementation on FPGA

It is said that deep learning, especially Recurrent Neural Network (RNN), is well-suited to FPGA implementation. This is because a RNN has the recurrent nature that can be easy for FPGA to parallelize and can be difficult for a CPU and a GPU. This paper [4] introduces the implementation of LSTM, which is a popular RNN, on FPGA because the GPU can not process enough in parallel due to the sequential component of the RNN. In their implementation, they separated the LSTM computation into three part, while sub processes in each three process are executed in parallel. Their results outperform the CPU implementation and the GPU implementation in terms of the execution time and the power efficiency. This results indicate that there are potential for the FPGA to accelerate deep learning than the GPU. Moreover, it also has an advantage of the power efficiency.

3.5 Multi-FPGA Implementation

When the size of the model increases, it gets more difficult to implement on single FPGA due to the limit of the on-chip memory. Also, when the datasets is too large, it takes huge time to finish the training on a single FPGA. Therefore it is necessary to implement on multiple FPGAs for a large model. Hence, many researchers have been working on a method of the implementation on multiple FPGAs. For instance, Dongup Kwon, Suyeon Hur *et al.* had introduced strategies of partitioning a large RNN model in order to accelerate the model using multiple FPGAs [12]. One of their strategies for partitioning is layer-wise partitioning. This strategy is for a model with a multiple layers of RNN. They also introduced a strategy called Row-wise partitioning which is used a single layer of RNN. The results show that Row-wise partitioning outperforms other simple partitioning methods such as Input-wise partitioning and Gate-wise partitioning. According to this paper, their strategies have great results in terms of the speedup of the inference time.

There is another research about the acceleration and load balancing of Convolutional Neural Network (CNN) using multiple FPGAs which is focused on training [8]. In this paper, it is explained that there are two types of methods for the implementation on multiple FPGAs: Data Parallelism and Model Parallelism. Data Parallelism is a method that duplicate the model in order to implement on multiple FPGAs, and separate the dataset into multiple groups so that large part of the dataset is process in parallel. It has a merit that can be processed in parallel with high-level, however, it has demerit that it requires a large on-chip memory. On the other hand, model parallelism is a method to divide the model into small part and implement that parts on each FPGA. Thus, each executions can be pipelined with the layers which leads to the better execution time.

Although it requires only small on-chip memory, the number of data transfer between the FPGAs is large which might be the bottleneck for the whole execution.

3.6 High Level Synthesis

High Level Synthesis (HLS) tools are widely known and used for a FPGA implementation rather than using Hardware Description Language (HDL). This is because the productivity and the scalability using HLS outperform HDL design flows. On the other hand, however, an accuracy of an application which is implemented by HLS tools are not very high as an implementation using HDL due to the overhead which is created when translating High Level Language (HLL) such as C/C++ to HDL. There is a paper which compares HLS design and HDL design for image processing [15]. This paper shows the result that HLS design has almost same pipeline length and utilization of used multipliers and slices as HDL design. Moreover, although there is an overhead for translating HLL to HDL which increases critical-path delays and needs extra sub-optimal design, the core frequency for HLS design and HDL design are relatively similar, which are 382 MHz and 428 MHz, respectively. From the result that HLS design does not let the frequency huge decrease compared to the HDL design, we think that we are able enough to use HLS tools to implement an application on FPGAs in terms of the execution time.

3.7 Virtualization and Cloud

The main difference between the virtualization and the cloud is that virtualization is a technique of dividing the function of server and network logically, while cloud is a service which users can freely use without preparing a software or data on-site. There is a virtualization research whose approach is to integrate virtualized FPGA-based hardware accelerators into commercial scale cloud computing systems [3]. Stuart Byma, J. Gregory Steffan, etc. have introduced a framework for multiple FPGAs in the generic cloud resources with OpenStack (open source cloud software) which enables users to use the same command for booting the regular Virtual Machine. According to this paper, their virtualization and abstraction reduced the design iteration time and the design complexity, and led to a better performance for the FPGA cloud compute resources than for virtual machines.

3.8 Research of FPGA using Cloud Service

Many companies such as Amazon, Google, Microsoft, Alibaba etc. are providing their own cloud services, which are Amazon Web Service (AWS), Google Cloud Platform (GCP), Azure, Alibaba Cloud, respectively, to people in the world. Each cloud services has computer system resources and often used for data storage (cloud storage). However, recently, they are also used for the acceleration for an application using devices such as CPU, GPU, FPGA, etc. on cloud. For example, AWS has instances with FPGAs called AWS EC2 F1 instance which are used for an acceleration.

There is a research about reducing energy consumption of CNN application using cloud multi-FPGA platform [16]. Multi-FPGA platform is often used for data center applications such as Deep Neural Network (DNN), big data processing, which can be easily parallelized. This is because the workloads of data center applications are not stable and vary each time which can usually leads to the waste of resources and power consumption. Cloud platform can optimize the use of the resources as well as the power consumption, and this paper introduced methods that how the instances are paralleled in each kernels. It also indicated a way to allocate to the FPGAs, and their method gives the number of FPGAs whose power are on and their clock frequencies. They discussed using the profile reports of FPGA from Xilinx SDAccel and AWS EC2 F1.x16large instance with Xilinx UltraScale+ FPGAs. They proposed a method to obtain the configuration bitstreams which are optimized by power efficiency for multiple FPGAs, and got results of better power consumption compared to their fastest implementation.

3.9 Research of Scheduling methods using Cloud Service

Many researchers have been working on how to minimize the execution time as well as the costs for using public cloud resources.
Stanislaw Deniziak and Slawomir Bak introduced a scheduling method to minimize the cost of resource usage and to minimize the latency of all activated applications [5]. Their method is for a HHPCaaS (heterogeneous HPSCaaS) clouds and they managed to reduce the costs by over 50% compared to the classical approach by sharing the resource between the applications.
Yihong Gao, Huadong Ma, Haito Zhang, Xiangqi Kong and Wangyang Wei considered the concurrency optimized task scheduling problem [7]. They proposed a delay constrained scheduling approach, which they use a approach to achieve the deadline, minimize the total costs for using the public cloud resource and minimize the total execution time that the cost can purchase, and a resource constrained scheduling approach, which they use a method to minimize the latest completion time and make sure the total costs for using the public cloud resource is lower and worthy. According to the results of these two approaches, their approaches can significantly improve the efficiency and simultaneously spend lower cost. Specially, the resource constrained scheduling approach got the lowest completion time while spending lower cost to achieve the goal.
There is another research using more complex model than a model we use in this paper. Sifei Lu, Xiaorong Li, Long Wang, Henry Kasim, Henry Palit, Terence Hung, Erika Fille Tupas Legara and Gary Lee used EC2 spot instances from AWS which fluctuate dynamically [13]. They proposed a dynamic resource provisioning solution to running large scalable application with hybrid instances includes both spot and on-demand instances. They used the on-demand instances for high priority tasks and used the spot instances for the normal computation tasks. Their method takes the dynamic pricing of cloud instances into consideration and it reduces the cost for using public cloud resources tolerates the failures for running large-scale applications. They used 1000 AWS EC2 instances and they obtained that their method achieved 23.3% cost savings

compared with the method using only on-demand instances, while the total completion time only increase 5.3% with backup fault tolerance policy.

3.10 Summary

In this chapter, we introduced researches about the FPGA implementation of DNN, including CNN and RNN. Also, we introduced a research about HLS in order to show HLS tools is efficient and does not lose the accuracy so much compared to HDL implementations. Moreover, we showed researches about using a cloud service as well as the virtualization. In addition, we also introduced three research about scheduling method for reducing the cloud usage fee as well as the execution time. As the vital part, We have introduced the related works that worked on estimating the function of DNA from the DNA sequences. These two deep learning model, DeepSEA and DanQ, have a great result of accuracy for estimation. Although DanQ has the more accuracy than DeepSEA, there is a problem that it takes massive time for training. Thus, in this paper, we propose a method to approach a solution to the problem.

Chapter 4

Multi-FPGA Implementation

4.1 Introduction

In this chapter, we introduce our proposed method and the implementation on multiple FPGAs. By separating the process on multiple FPGAs the way they can process individually, we can obtain faster training time.
We also indicate the result we obtained by implementing the DanQ model on FPGAs. We show three different implementations that are implemented on single FPGA, dual FPGA and 8 FPGAs. The details are explained in the following sections.

4.2 Target Model

The target model we chose is the DanQ model shown in Figure 3.2. The output feature map size, the input feature map size and the filter size of each layer are shown in Table 4.1. DanQ is a hybrid model with CNN and RNN. Although it has an outstanding performance in accuracy, its training time is enormous.

Table 4.1: The Size of Feature Map of Input and Output of each layer

Name of the Layer	Input Map Size	Output Map Size	Filter Size
Convolutional Layer	1000×4	975×320	$4 \times 320 \times 26$
ReLU Layer	975×320	975×320	-
Max Pooling Layer	975×320	75×320	-
Dropout Layer	75×320	75×320	-
BiLSTM Layer	75×320	75×640	$320 \times 320 \times 4 \times 2 \times 2$
Dropout Layer	75×640	75×640	-
Full-connected Layer	75×640	925	$75 \times 640 \times 925$
ReLU Layer	925	925	-
Full-connected Layer	925	919	925×919
Sigmoid Layer	919	919	-

4.3 Proposed Method for FPGA Implementation

There is much research about Genome analysis using FPGA. FPGA has an advantage of processing small bit-width like DNA sequence, which can be represented as 2 bit for bit-width. As we mentioned in Chapter 3, the training for DanQ [14] requires a huge dataset of 4,440,000, with each one represents the DNA sequence whose length is 1000. According to the paper [14], the training for one epoch requires almost one hour. Therefore, we think it is sufficient to distribute the huge data to multiple FPGA, which can lead to the training acceleration. Moreover, we focused on the BiLSTM layer, which consumes about 48% of the whole training time. BiLSTM is time-consuming because there are two LSTM to process. Also, a BiLSTM layer is relatively easy to divide the process which leads to the availability of parallel processing. By dividing these two BiLSTM process into two parts, the processing time can be decreased: the LSTM which reads the input forward is implemented on an FPGA while the other LSTM which reads the input backward is implemented on another FPGA. In addition, we process the BiLSTM layer in parallel using 8 FPGAs by dividing each LSTM layers into 4 parts independently.

4.4 The Experimental Environment for AWS

For implementing our application on FPGA, we used an instance from Amazon Web Service, and the detail of the instance is shown in Table 4.2.

Table 4.2: The detail of Xilinx UltraScale+ VU9P

	VU9P
System Logic Cells (K)	2,586
CLB Flip-Flops (K)	2,364
CLB LUTs (K)	1,182
Max. Dist. RAM (Mb)	36.1
Total Block RAM (Mb)	75.9
URAM (Mb)	270
DSP slices	6,840
Peak INT8 DSP(TOP/s)	21.3
PCIe Gen3 x16	6
Max. Single-Ended HD I/Os	832
GTY 32.75Gb/s Transceivers	120

The F1 Instance is a compute instance with FPGA, which we can program to create

custom hardware accelerations for our application. Xilinx UltraScale+ VU9P FPGA is on the F1 Instance, and the main advantages are described below:

- High-density device fabricated in a 16nm process
- 2.5 million logic cells
- More than 6800 DSP engines
- Four DDR4 channels
- Bandwidth 48 GB/s
- Massive I/O transfers
- Massive parallelism

One of the big difference to other FPGA such as Intel FPGA is an on-chip memory on the device. For Xilinx UltraScale+ FPGAs, there are not only Block Random Access Memory (BRAM) but also the Ultra Random Access Memory (URAM). The URAM is a new block memory for Xilinx UltraScale+ family and has totally 500 MB on-chip storage at the maximum. One block for the URAM is much bigger than that of BRAM. Hence, it is suitable for relatively large variables to store in the URAM and it is not a smart way to use the URAM for many small variables. Thus, a better design for an implementation is to store large variables on the URAM and store small variables on the BRAM.

In AWS, there are three types of F1 Instances, and the details of them are shown in Table 4.3. Each instance has one or more FPGAs, and the on-demand price for using an instance increases as the size of the instance gets large.

Table 4.3: Types of F1 Instances

Name	**FPGAs**	**vCPUs**	**Instance Memory (GiB)**	**SSD Storage (GB)**	**On-Demand Price/hr**
f1.2xlarge	1	8	122	470	$ 1.65
f1.4xlarge	2	16	244	940	& 3.30
f1.16xlarge	8	64	976	4×940	$ 13.20

We mainly used SDAccel and Vivado HLS as a tool to do my experiment. SDAccel is an Integrated Development Environment (IDE) for the implementation on actual FPGA. Vivado HLS is a tool for generating Register Transfer Level (RTL) codes from C/C++ codes using High-Level Synthesis (HLS).

Table 4.4: Detail of Used Tools

Tools	Detail
SDAccel 2019.1	Integratd Development Environment for FPGA Implementation
Vivado HLS 2019.1	High-Level Synthesis tool for generating RTL from C/C++

Table 4.5: Specifications of Resources

Resources	Specification
CPU	Intel(R) Xeon(R) CPU E5-2686 v4 @ 2.30GHz
FPGA	Xilinx UltraScale+ VU9P

4.5 Implementation

In order to implement DanQ model in Figure 3.2, which I mentioned in chapter 3, on FPGA using HLS, I programmed the model by C++ without using a framework for deep learning.

4.5.1 Quantization

We have five parameters that can be quantized: weight, activation, gradient, error and temporary values, which is used for RMSprop. It is obvious that the more bit-width the parameters have, the more range of values they can represent. Thus, it is more likely to have better training if these parameters have a larger bit-width. However, it is not easy to know the most appropriate bit-width for each of them to have the highest training accuracy. Therefore, in this paper, we decided to have these five parameters the same bit-width of 16, which satisfy the limit of hardware utilization in on-chip memory in FPGA.

4.5.2 Hardware Utilization

We first analyzed the hardware utilization for the BiLSTM layer by getting an HLS report from Hardware emulation in SDAccel environment, which checks the correctness

of the logic generated for the compute units and tests the functionality of the logic that will be executed on the FPGA.

The table4.6 shows the estimation of hardware utilization of BRAM, DSP Block, FF, LUT and URAM for Xilinx UltraScale+ VU9P from the hardware emulation of Vivado HLS. Here, Utilization SLR represents a percentage of utilization in one SLR. Since there are 3 SLRs in Xilinx UltraScale+ VU9P, we can accept until 300% of utilization SLR by dividing the kernel into three kernels. In our implementations, we used URAM blocks for weights and gradients and used BRAM blocks for the other parameters such as activations and Errors.

Table 4.6: Hardware Utilization of single FPGA implementation

	BRAM	**DSP**	**FF**	**LUT**	**URAM**
Utilization (%)	101	9	6	10	83
Utilization SLR(%)	303	28	20	31	250

According to the table 4.6, the utilization of BRAM is 101%, which is exceeding the capacity of BRAM in Xilinx UltraScale+ VU9P. Therefore, we need the quantization to decrease the size of parameters such as weight and activation to make BRAM utilization smaller than the maximum utilization of Xilinx UltraScale+ VU9P.

Table 4.7 shows the estimation of hardware utilization from the hardware emulation of Vivado HLS after the uniform quantization. In this research, we quantized the parameters such as weights, activations, gradients and errors to the bit-width of 16 in order to satisfy the requirement of harware utilization of Xilinx UltraScale+ VU9P. We decided the bit-width of 16 from the perspective of the hardware usage. We found out that we could not implement the model using the parameters with the fixed-point number of 32 bits because of the hardware limitation. Thus, we chose to quantize the parameters to the fixed-point number of 16 bits.

Table 4.7: Hardware Utilization of single FPGA implementation

	BRAM	**DSP**	**FF**	**LUT**	**URAM**
Utilization (%)	53	3	10	24	83
Utilization SLR(%)	160	11	31	72	250

According to this Table 4.7, BRAM utilization is 53%, which is under the maximum BRAM capacity for Xilinx UltraScale+ VU9P.

4.5.3 Implementation for Single FPGA

Algorithm 1 shows the algorithm of the host implementation for a single FPGA. We have two times of booting the kernel for both forward and backward processes, shown in Line $6-9$ and Line $16-19$, respectively. We call standard OpenCL API to interact with the FPGA-accelerated functions. OpenCL is a framework for writing programs for FPGAs. The loss calculation is executed in the host to output the loss value, which is shown inline 13. Line 24 represents the updating process of weights using the parameter h, which is used for the optimizer RMSprop [9].

Algorithm 2 shows the algorithm of kernel implementation. We used a switch statement to process two types of processes: forward process and backward process. These processes are shown inline 5 and 10. Line 13 represents the updating process of weights in kernel, which is the same as the updating process in the host.

4.5.4 Implementation for Dual FPGA

We propose a method to implement the BiLSTM layer on dual FPGA which is shown in Figure 4.1.

Algorithm 1 *Host Implementation on Single FPGA*

```
1: for count ← 1 to epochs + 1 do
2:   for it ← 1 to noumber of iterations do
3:     input = load_data(dataset_in)
4:     correct_output = load_data(dataset_ans)
5:     datax = predict1(input, buffer)
6:     OCL_CHECK(err, cl :: CommandQueueq_forward(context, device,
       CL_QUEUE_PROFILING_ENABLE, &err))
7:     OCL_CHECK(err, err = q_forward.enqueueMigrateMemObjects
       (datax, 0))
8:     OCL_CHECK(err, err = q_forward.enqueueTask(kernel))
9:     OCL_CHECK(err, err = q_forward.enqueueMigrateMemObjects
       (datay, CL_MIGRATE_MEM_OBJECT_HOST))
10:    q_forward.finish()
11:    bufferx = datay
12:    datax = predict2(bufferx, buffery)
13:    l = loss.forward(buffery, correct_output)
14:    dout = loss.backward(1)
15:    dout = gradient1(dout, buffery)
16:    OCL_CHECK(err, cl :: CommandQueueq_backward(context, device,
       CL_QUEUE_PROFILING_ENABLE, &err))
17:    OCL_CHECK(err, err = q_backward.enqueueMigrateMemObjects
       (dout, 0))
18:    OCL_CHECK(err, err = q_backward.enqueueTask(kernel))
19:    OCL_CHECK(err, err = q_backward.enqueueMigrateMemObjects
       (dout_output, CL_MIGRATE_MEM_OBJECT_HOST))
20:    q_backward.finish()
21:    buffero = dout_output
22:    final = gradient2(buffero)
23:    if count==100 then
24:      weight = update(grads, h)
25:      grads = 0
26:    end if
27:  end for
28: end for
```

Algorithm 2 *Kernel Implementation*

switch (θ)
case f**:**
  // Forward Process in Kernel
  $kernelx = datax$
  $kernely = predict\_kernel(kernelx)$
  $datay = kernely$
case b**:**
  // Backward Process in Kernel
  $kerneld = dout$
  $kerneld = backward\_kernel(kerneld)$
  $dout\_output = kerneld$
  $weights\_BiLSTM = update(grads\_BiLSTM, h\_BiLSTM)$
  $grads\_BiLSTM = 0$
end switch

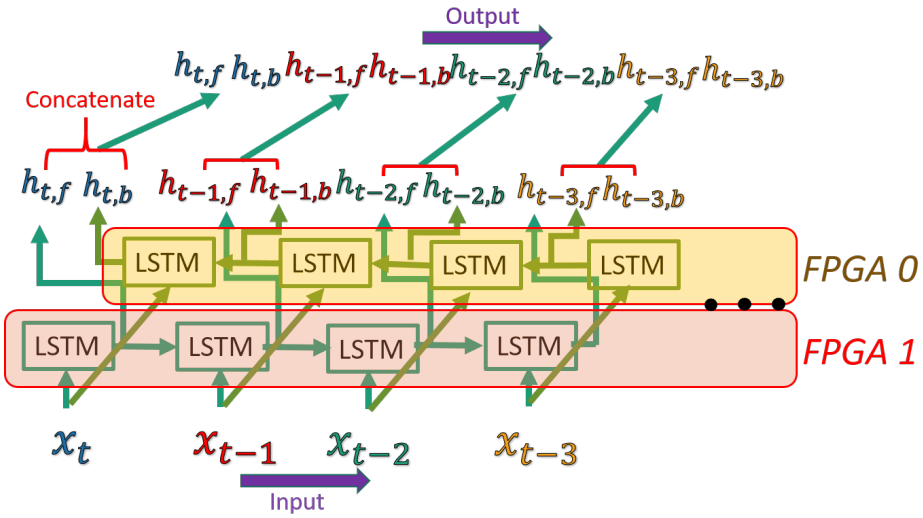

Figure 4.1: The overview Double-FPGA Implementation

Algorithm 3 shows the algorithm of the host implementation for dual FPGA. The difference between the single FPGA implementation shown in Algorithm 1 and dual FPGA implementation shown in Algorithm 3 is that dual FPGA implementation requires two different CommandQueue for two different FPGAs, which are shown in Line 2 and Line 3. We created two inputs that both values of inputs are different orders and send them to the FPGAs, respectively. After the executions on both FPGAs are finished, the host receives the two different outputs from the two FPGAs and concatenates them in the correct order, as shown in Line 23. In the backward process phase, the host sends two different errors to the FPGAs, where an error sent to an FPGA has to correspond to the activation, which is output from the same FPGA, and then the FPGAs send errors back to the host. In the end, the host sums up the error sent from both FPGAs and continues the backward process, which are shown in Line 40 and Line 41.

4.5.5 Implementation for 8 FPGAs

We proposed the implementation methods of BiLSTM on 8 FPGAs. We took the row-wise partitioning strategy introduced by [12] into consideration. The way we separated the BiLSTM into two parts, as mentioned above, is also used in this implementation. The vital part of this idea is that we divide the activation into four blocks in order to proceed in parallel, which we call it Block-wise Implementation shown in 4.2. These processes can be processed independently so that this block-wise implementation can outperform other implementation in terms of the execution time.

Algorithm 3 *Host Implementation on Dual FPGA*

```
1: for d ← 1 to device_count do
2:     OCL_CHECK(err, q_forward[d] = cl :: CommandQueue
       (context[d], device[d], CL_QUEUE_PROFILING_ENABLE, &err))
3:     OCL_CHECK(err, q_backward[d] = cl :: CommandQueue
       (context[d], device[d], CL_QUEUE_PROFILING_ENABLE, &err))
4: end for
5: for count ← 1 to epochs + 1 do
6:     for it ← 1 to noumber of iterations do
7:         input = load_data(dataset_in)
8:         correct_output = load_data(dataset_ans)
9:         datax = predict1(input, buffer)
10:        OCL_CHECK(err, cl :: CommandQueueq_forward[0]
           (context, device, CL_QUEUE_PROFILING_ENABLE, &err))
11:        OCL_CHECK(err, cl :: CommandQueueq_forward[1]
           (context, device, CL_QUEUE_PROFILING_ENABLE, &err))
12:        OCL_CHECK(err, err = q_forward[0].enqueueMigrateMemObjects(datax[0], 0))
13:        OCL_CHECK(err, err = q_forward[1].enqueueMigrateMemObjects(datax[1], 0))
14:        OCL_CHECK(err, err = q_forward[0].enqueueTask(kernel[0], NULL, &events[0]))
15:        OCL_CHECK(err, err = q_forward[1].enqueueTask(kernel[1], NULL, &events[1]))
16:        OCL_CHECK(err, err = q_forward[0].enqueueMigrateMemObjects
           (datay[0], CL_MIGRATE_MEM_OBJECT_HOST))
17:        OCL_CHECK(err, err = q_forward[1].enqueueMigrateMemObjects
           (datay[1], CL_MIGRATE_MEM_OBJECT_HOST))
18:        for auto queue : q_forward do
19:            OCL_CHECK(err, err = queue.flush()
20:            OCL_CHECK(err, err = queue.finish()
21:        end for
22:        q_forward.finish()
23:        buffx = datay  // datay is concatenated with datay[0] and datay[1]
24:        datax = predict2(buffx, buffy)
25:        l = loss.forward(buffy, correct_output)
26:        dout = loss.backward(1)
27:        dout = gradient1(dout, buffy)
28:        OCL_CHECK(err, cl :: CommandQueueq_backward[0]
           (context, device, CL_QUEUE_PROFILING_ENABLE, &err))
29:        OCL_CHECK(err, cl :: CommandQueueq_backward[1]
           (context, device, CL_QUEUE_PROFILING_ENABLE, &err))
30:        OCL_CHECK(err, err = q_backward[0].enqueueMigrateMemObjects(dout[0], 0))
31:        OCL_CHECK(err, err = q_backward[1].enqueueMigrateMemObjects(dout[1], 0))
32:        OCL_CHECK(err, err = q_backwar[0].enqueueTask(kernel[0]))
33:        OCL_CHECK(err, err = q_backwar[1].enqueueTask(kernel[1]))
34:        OCL_CHECK(err, err = q_backward[0].enqueueMigrateMemObjects
           (dout_output[0], CL_MIGRATE_MEM_OBJECT_HOST))
35:        OCL_CHECK(err, err = q_backward.enqueueMigrateMemObjects
           (dout_output[1], CL_MIGRATE_MEM_OBJECT_HOST))
36:        for auto queue : q_backward do
37:            OCL_CHECK(err, err = queue.flush()
38:            OCL_CHECK(err, err = queue.finish()
39:        end for
40:        buffero = dout_output[0] + dout_output[1]
41:        final = gradient2(buffero)
42:        if count==100 then
43:            weight = update(grads, h)
44:            grads = 0
45:        end if
46:    end for
47: end for
```

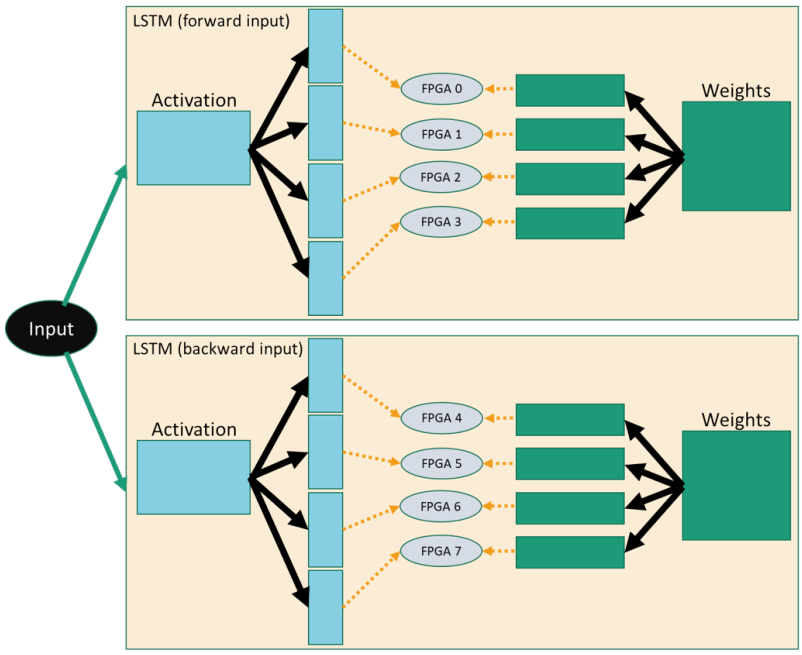

Figure 4.2: The overview of Double-FPGA Implementation

The algorithm of this implementation, shown in Algorithm 4, is similar to that of dual FPGA. The input is read in different directions, like dual FPGA version, to two groups of four FPGAs, and each input is separated into four parts to input to the four FPGAs inside its group, which are shown in Line 10 to Line 13. The backward process is just as same as the forward process. This implementation requires eight different CommandQueue, which are shown in Line 17 to Line 24 and Line 42 to Line 49, and simultaneously executed after all the queues needed for the execution are set, which are shown in Line 25 to Line 28 and Line 50 to Line 53.

4.6 Experiments

The implementation that we did is the DanQ model shown in 4.1. We focused on the BiLSTM layer in DanQ model and implemented it on AWS EC2 F1 Instance for training. Layers other than the BiLSTM layer are implemented on a CPU. The overview of implementation on a CPU and an FPGA is shown in 4.3. There are four times transfers between the CPU and the FPGA. First, there is a forward process for four layers (convolutional layer, ReLU layer, Max Pooling layer and Drop out layer) on the CPU and send the activation to the FPGA. When the FPGA gets the activation, the forward process for the BiLSTM layers is executed and returns the activation to the

Table 4.8: Comparison of execution time between CPU and FPGA

Resources (# of FPGAs)	**BiLSTM Layer (1 input)**			All Layers (1 epoch)
	Forward (ms)	Backward (ms)	Total (ms)	Whole Time (s)
CPU	441.635	755.445	1197.08	1278.64
Single-FPGA	387.815	750.050	1137.865	1161.16
Dual-FPGA	195.274	376.587	571.861	885.340
8 FPGAs	54.371	145.194	199.565	695.914

CPU. The CPU does the forward process for the rest of the layers (Drop out layer, Full-connected layer, ReLU layer, Full-connected layer and Sigmoid layer) and calculate the loss value using binary cross-entropy. After that, the backward process is executed for the five layers (Drop out layer, Full-connected layer, ReLU layer, Full-connected layer and Sigmoid layer) and send the error to the FPGA. The FPGA executes the backward process for the BiLSTM layer and returns the error to the CPU to let it execute the backward process for the four layers (convolutional layer, ReLU layer, Max Pooling layer and Drop out layer).

We also implemented the CPU version, which implemented the whole DanQ layer on the CPU. We compared the CPU version and the FPGA version in terms of their execution time and their loss value.

4.7 Results

The result of comparison between the CPU version and FPGA version in terms of their execution time and their loss are shown in Table 4.8 and Figure 4.4, respectively. Moreover, we also compared three FPGA implemetations: Single-FPGA implementation, Dual-FPGA implementation and 8 FPGA implementation. Table 4.8 shows the execution time of the BiLSTM layer including the transfer time between the host and the kernel in a case of having 1 input data. This is not including the update process as well as the transfer time of the parameters that are needed to continue the training with a different instance. In addition, Table 4.8 also indicates the execution time of all layers for 1 epoch which has 500 input data. This results are used in our second proposed method that are introduced in Chapter 5.

Although the execution time for a single FPGA implementation is slightly better than that of CPU by 1.05x, the execution time of dual-FPGA implementation is much better than that of the CPU and that of the single-FPGA implementation. The execution time for dual-FPGA implementation is 1.99x faster than that of the single-FPGA implementation. Moreover, the implementation for 8 FPGAs outperforms the single-FPGA implementation and the CPU implementation by 5.70x and 6.00x, respectively, in terms of their execution time. Also, the execution time for 8 FPGAs implementation is 2.87x faster than that of the dual-FPGA implementation. This is because all the processes of the BiLSTM layer are executed in parallel on multiple FPGAs which led to faster

execution.

As shown in Figure 4.4, we obtained almost the same training loss in FPGA implementation compared to the CPU implementation even if we quantized the parameters when we implement on FPGAs. Therefore, there is no problem for using the uniform quantization for the model. From this result, we think that this model does not require the large bit width because the input data are one hot vectors which can be represented by the bit width of two.

4.8 Discussion

We noticed that the size of the DanQ model is too large to implement on a single FPGA, so that it is hard to accelerate by a technique of data parallelism using multi-FPGA. We obtained that we need to use a technique of model parallelism in order to implement the whole DanQ model on FPGAs. However, this cannot be efficient because there can be much transfer time between the FPGAs, leading to higher latency. Therefore, we only focused on a BiLSTM layer which consumes the large part of the whole training. We proposed the methods for dual-FPGA implementation and 8 FPGAs implementation for the BiLSTM layer, and these methods are specialized for a single BiLSTM layer which has fewer overhead for data transfer compared to multiple BiLSTM layers.

4.9 Summary

This chapter indicated our proposed method and the way we implemented them. We have three implementations for three different sizes of instances. To implement the BiLSTM layer on FPGAs, we need to use uniform quantization to the bit width of 16 in order to reduce the model size. We divided the BiLSTM layer into two parts for the dual FPGA implementation, which we consider as two LSTM layers, and implemented them on each FPGA. On the other hand, 8 FPGAs implementation has the same idea as that of dual FPGA implementation but divides the LSTM layer into four parts in order to process in parallel of 8.

We also showed the results of implementing the DanQ on FPGAs and compared them with a CPU in terms of the execution time and the loss tendency. We could accelerate the DanQ model by using a single FPGA by 1.05x compared to our CPU implementation without losing the training accuracy. Besides, our implementation on 8 FPGAs gets 2.87x faster than the dual FPGA implementation and 6.00x faster than the CPU implementation.

Algorithm 4 *Host Implementation on* 8 *FPGAs*

```
for d ← 0 to device_count do
    OCL_CHECK(err, q_forward[d] = cl::CommandQueue
    (context[d], device[d], CL_QUEUE_PROFILING_ENABLE, &err))
    OCL_CHECK(err, q_backward[d] = cl::CommandQueue
    (context[d], device[d], CL_QUEUE_PROFILING_ENABLE, &err))
end for
for count ← 1 to epochs + 1 do
    for it ← 1 to noumber of iterations + 1 do
        input = load_data(dataset_in)
        correct_output = load_data(dataset_ans)
        datax = predict1(input, buffer)
        for d ← 0 to device_count/2 step 2 do
            dataxd[d] = buffer[d * sizeof datax/4]
            dataxd[d+1] = buffer[d * sizeof datax/4]
        end for
        for d ← 0 to device_count do
            OCL_CHECK(err, cl::CommandQueueq_forward[[d]]
            (context[d], device[d], CL_QUEUE_PROFILING_ENABLE, &err))
        end for
        for d ← 0 to device_count/2 step 2 do
            OCL_CHECK(err, err = q_forward[d].enqueueMigrateMemObjects(dataxd[d], 0))
            OCL_CHECK(err, err = q_forward[d+1].enqueueMigrateMemObjects(dataxd[d+1], 0))
            OCL_CHECK(err, err = q_forward[d].enqueueTask(kernel[d], NULL, &events[d]))
            OCL_CHECK(err, err = q_forward[d+1].enqueueTask(kernel[d+1], NULL, &events[d+1]))
            OCL_CHECK(err, err = q_forward[d].enqueueMigrateMemObjects
            (datay[d], CL_MIGRATE_MEM_OBJECT_HOST))
            OCL_CHECK(err, err = q_forward[d+1].enqueueMigrateMemObjects
            (datay[d+1], CL_MIGRATE_MEM_OBJECT_HOST))
        end for
        for auto queue : q_forward do
            OCL_CHECK(err, err = queue.flush())
            OCL_CHECK(err, err = queue.finish())
        end for
        q_forward.finish()
        buffx = datay
        buffy = backward(buffx)
        l = loss.forward(buffy, correct_output)
        dout = loss.backward(1)
        dout = gradient1(dout, buffy)
        for d ← 0 to device_count/2 do
            doutd[d] = dout[d * sizeof datax/4]
            doutd[d+1] = dout[d * sizeof datax/4]
        end for
        for d ← 0 to device_count do
            OCL_CHECK(err, cl::CommandQueueq_backward[[d]]
            (context[d], device[d], CL_QUEUE_PROFILING_ENABLE, &err))
        end for
        for d ← 0 to device_count/2 step 2 do
            OCL_CHECK(err, err = q_backward[d].enqueueMigrateMemObjects(doutd[d], 0))
            OCL_CHECK(err, err = q_backward[d+1].enqueueMigrateMemObjects(doutd[d+1], 0))
            OCL_CHECK(err, err = q_backward[d].enqueueTask(kernel[d], NULL, &events[d]))
            OCL_CHECK(err, err = q_backward[d+1].enqueueTask(kernel[d+1], NULL, &events[d+1]))
            OCL_CHECK(err, err = q_backward[d].enqueueMigrateMemObjects
            (dout_output[d], CL_MIGRATE_MEM_OBJECT_HOST))
            OCL_CHECK(err, err = q_backward[d+1].enqueueMigrateMemObjects
            (dout_output[d+1], CL_MIGRATE_MEM_OBJECT_HOST))
        end for
        for auto queue : q_backward do
            OCL_CHECK(err, err = queue.flush())
            OCL_CHECK(err, err = queue.finish())
        end for
        buffero = dout_output
        final = gradient2(buffero)
        if count==100 then
            weight = update(grads, h)
            grads = 0
        end if
    end for
end for
```

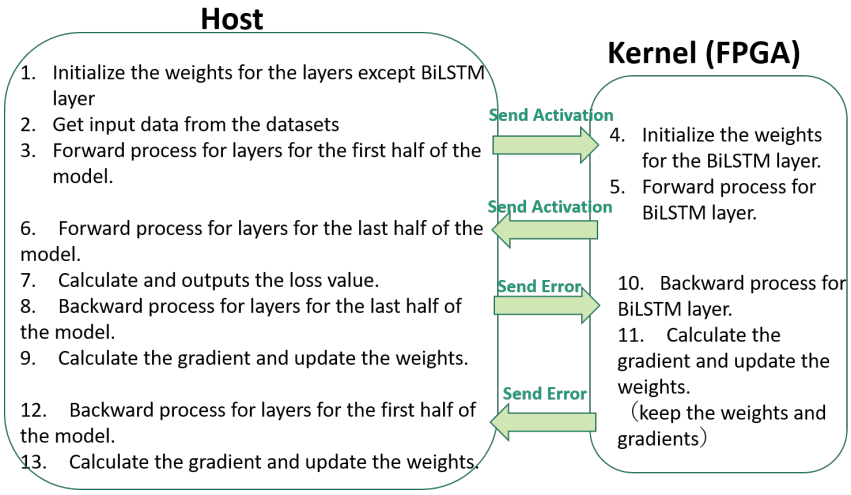

Figure 4.3: The Overview of the Implementation for Single FPGA

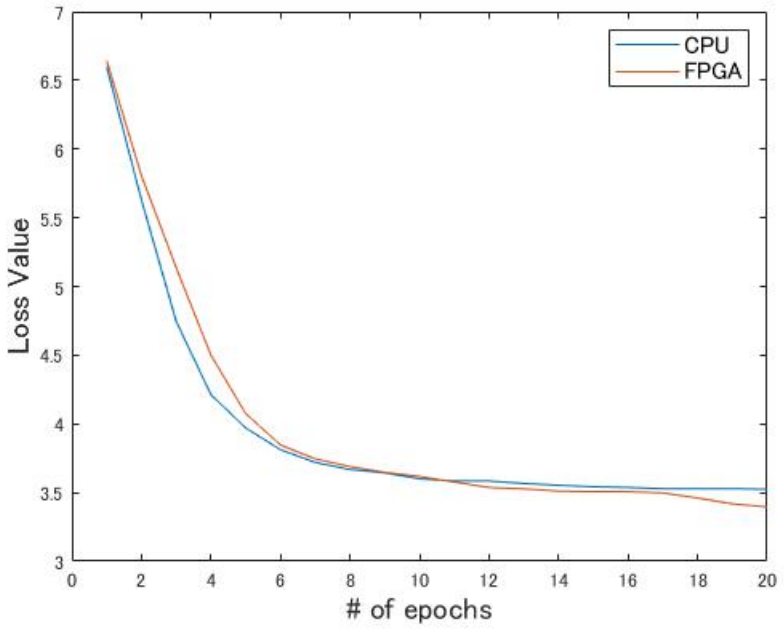

Figure 4.4: The comparison of loss tendency between CPU version and FPGA version

Chapter 5

Cloud Optimization

5.1 Introduction

In this chapter, we introduce our second proposed method and its evaluation. We provide an optimized training time and cloud usage fee to each cloud user by changing the resource size during the training depending on the cloud usage fee at that time. This is realized by saving and loading the parameters which are needed to continue training on a different resource.
Also, we show the results with our idea to optimize cloud resources depending on the cloud instance usage fee at that time. We use examples of optimizing the training time and the cloud instance usage fee with the user's need. Research about the scheduling methods which we showed in Chapter 3 are using CPU or heterogeneous devices. In this paper, we propose a method of optimization using FPGAs in a public cloud with targeting the BiLSTM layer in DanQ. The details are explained in the following sections.

5.2 Proposed Method for Cloud Optimization

Our implementation can change the instance size by saving the parameters which are needed to continue the training, such as weights. This enables to control the training time with the instance usage fee. Thus, the main focus of our research is to optimize the usage fee for the cloud instance. We propose a system to optimize the usage fee depending on the training time given by users. The usage fee fluctuates each time for using an instance in real cloud services such as AWS. Therefore, we introduce to change the instance size according to the usage fee at that time. Some users want to finish the training very early, no matter how much the usage fee costs, while others want to save the costs and can wait for a long time to finish the training. Hence, we switch the instance size during the training depending on the instance usage fee at the time so that we can train the model using the cloud services with the user-optimized instance usage fee.

5.3 A model for optimizing the instance usage fee

We define a model for optimizing the instance usage fee with the given training time from users. When we use an instance on cloud services, the usage fee always changes depending on the other cloud users. Basically, there is a big difference in usage fee between day time and night time. Figure 5.1 shows an example of the fluctuation between day and time. We discuss our optimization using this figure.

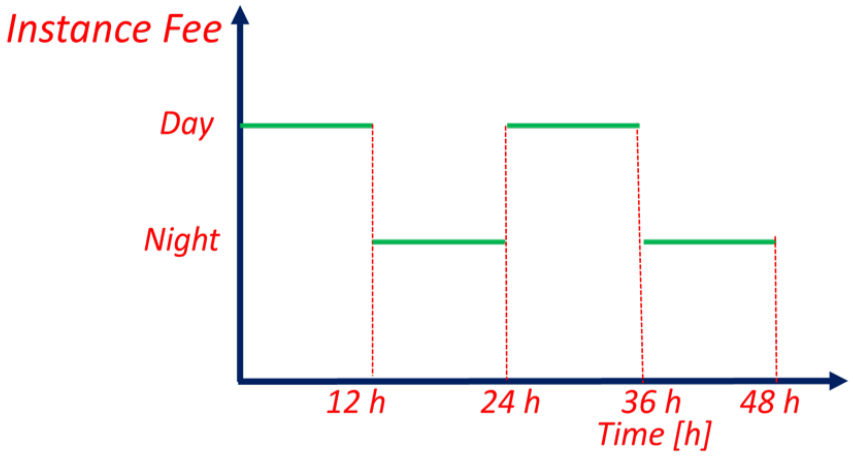

Figure 5.1: An example of the fluctuation of instance usage fee

5.4 Experiments

We used AWS as a cloud service to implement on multi-FPGA. We used an instance called AWS EC2 F1 Instance, which has FPGAs: f1.2xlarge for single FPGA implementation, f1.4xlarge for dual FPGA implementation, and f1.16xlarge for 8 FPGAs implementation. When we use AWS for development, we use AWS EC2 M4 Instance, which does not have FPGAs, for debugging our program as well as software emulation and hardware emulation. Software emulation is a CPU-based simulation that both the kernel code and the FPGA binary code (kernel code) are compiled to run on an x86 processor. This enables developers to iterate and refine the algorithms through fast compilation. On the other hand, Hardware emulation enables the developer to check the correctness of the logic generated for the FPGA binary. This invokes the hardware simulator in the environment to test the functionality of the code that will be executed

on the FPGA Custom Logic. In addition, we build the host application and FPGA binary using M4 Instance. At the end of using the instance, we generate an Amazon FPGA Image (AFI) by creating an AWS FPGA binary file (*.awsxclbin) from an FPGA binary file (*.xclbin) which is generated in the phase of building the host application and FPGA binary. The flow of the development using AWS is shown in Figure 5.2.

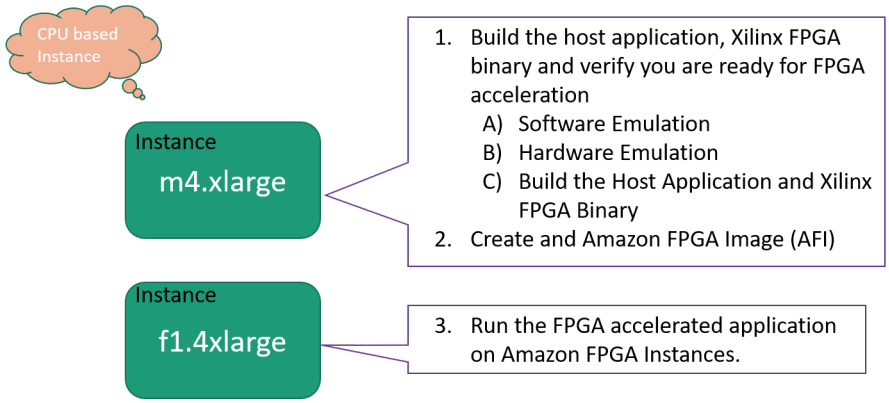

Figure 5.2: The Flow of Development using AWS

After the AFI is created, we start an FPGA instance on which we run the FPGA accelerated application. Then we copy the compiled host executable (exe) and AWS FPGA binary file to the instance. Finally, we execute the host application using the both compiled host executable and AWS FPGA binary file on the instance.
The overview of the HLS flow is shown in Figure 5.3.

We implemented on AWS EC2 F1 Instance called f1.4xlarge, which has two FPGAs, and the overview of the implementation is shown in Figure 5.4 We

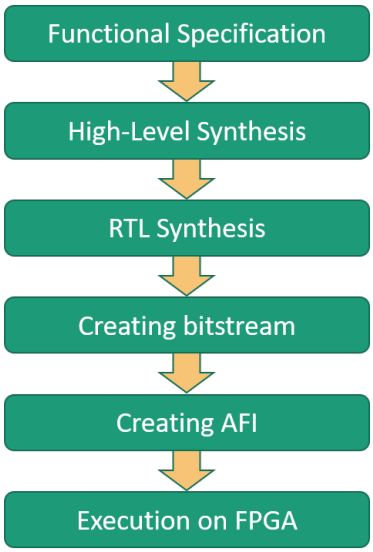

Figure 5.3: The Flow of using HLS

Table 5.1: Definition of the instance fee

Resource	**Time**	**Instance Usage Fee [$]**
Single-FPGA	Day	1.65
	Night	0.49
Dual-FPGA	Day	3.30
	Night	0.99
8 FPGAs	Day	13.20
	Night	3.96

5.5 Results

We trained the whole model of DanQ with 500 data for 150 epochs. In order to show an example of optimization, we assume the instance usage fee as Table 5.1. For the instance usage fee in the daytime, the costs shown in the table are exactly the same value as the on-demand instance usage fee in AWS. On the other hand, for the instance usage fee in the nighttime, the costs are the same as that of the spot instance in AWS, whose costs change depends on the number of cloud users.

Figure 5.5 shows an example of using f1.2xlarge instance (single-FPGA) in AWS with the case of fluctuation of instance usage fee shown in 5.1. When we use this instance for the whole time, the training time is 48.38 (h), and the instance usage fee is $ 51.79.

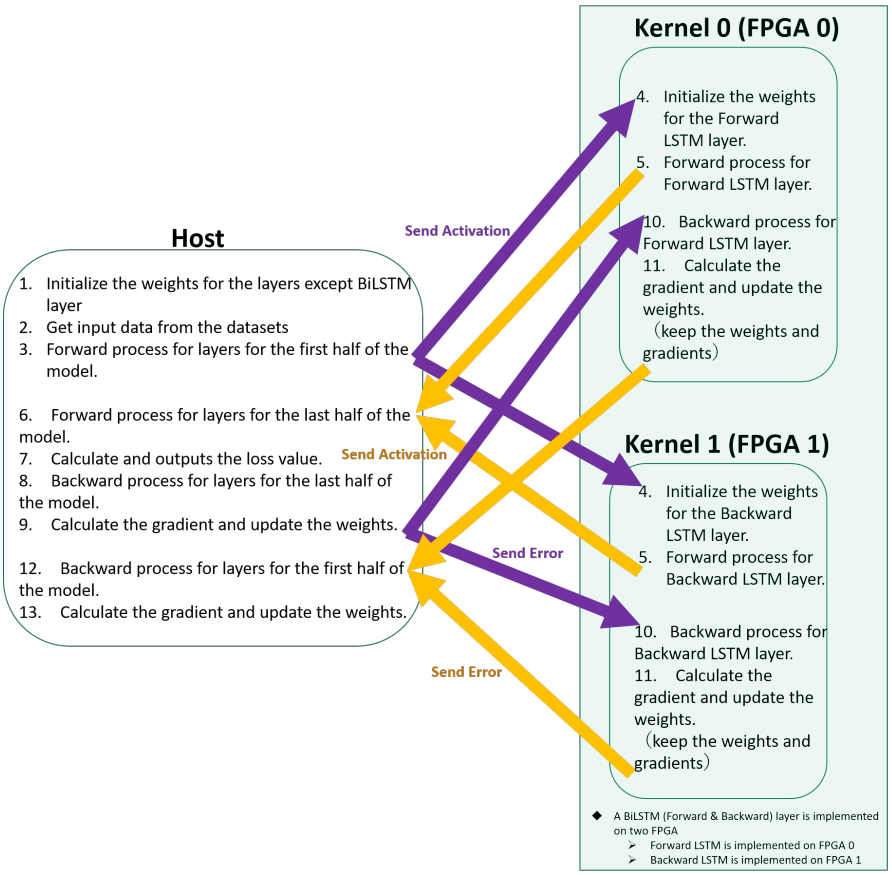

Figure 5.4: The Overview of the Implementation for the Dual FPGA

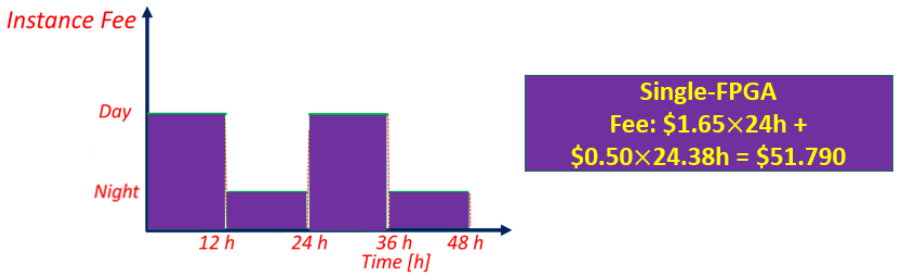

Figure 5.5: An example of using single FPGA for all time

Figure 5.6 shows an example of using f1.4xlarge instance (dual-FPGA) in AWS with the case of fluctuation of instance usage fee shown in 5.1. When we use this instance

for the whole time, the training time is 36.89 (h), and the instance usage fee is $ 91.96.

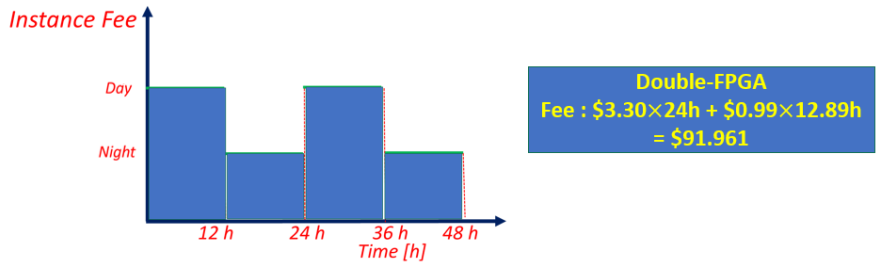

Figure 5.6: An example of using dual FPGA for all time

Figure 5.7 shows an example of using f1.16xlarge instance (8 FPGAs) in AWS with the case of fluctuation of instance usage fee shown in 5.1. When we use this instance for the whole time, the training time is 29.00 (h), and the instance usage fee is $ 271.92.

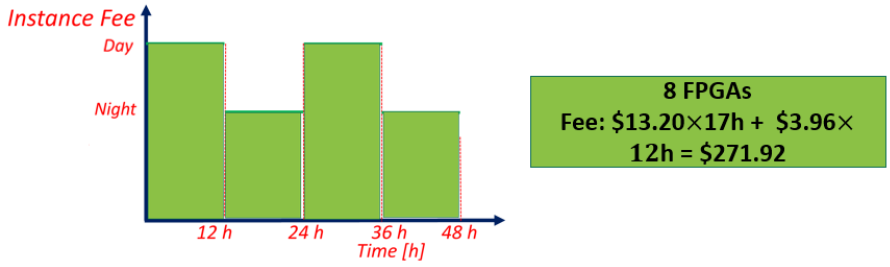

Figure 5.8: An example of using 8 FPGAs for all time

Figure 5.8 shows an example of changing the instance between f1.2xlarge and f1.4xlarge with the case of fluctuation of instance usage fee shown in 5.1. When we use the f1.2xlarge at daytime and f1.4xlarge at nighttime, the training time is 42.591 (h), and the instance usage fee is $ 58.01.

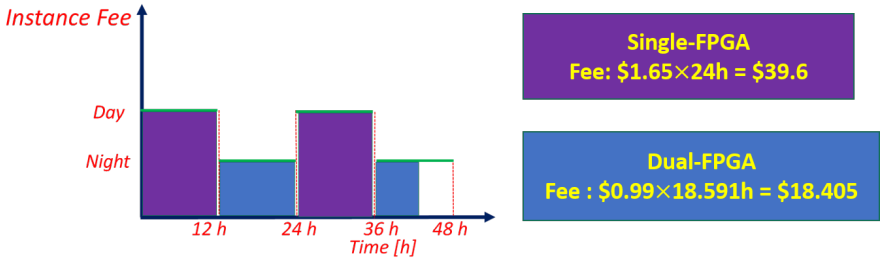

Figure 5.8: An example of optimizing the resource size between single FPGA and dual FPGA

Figure 5.9 shows an example of changing the instance between f1.2xlarge and f1.16xlarge with the case of fluctuation of instance usage fee shown in 5.1. When we use the f1.2xlarge at daytime and f1.16xlarge at nighttime, the training time is 38.61 (h), and the instance usage fee is $ 77.66.

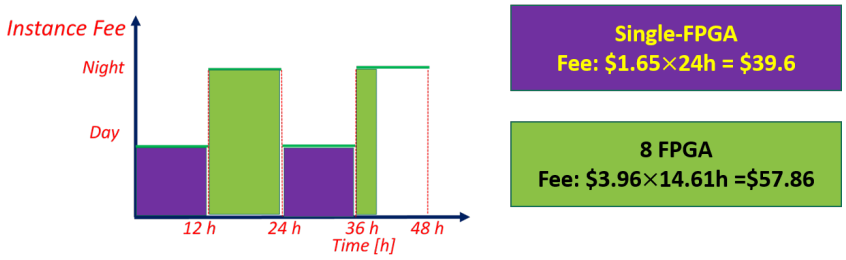

Figure 5.9: An example of optimizing the resource size between single FPGA and 8 FPGAs

Figure 5.10 shows an example of changing the instance between f1.4xlarge and f1.16xlarge with the case of fluctuation of instance usage fee shown in 5.1. When we use the f1.4xlarge at daytime and f1.16xlarge at nighttime, the training time is 33.63 (h), and the instance usage fee is $ 118.89.

According to the Figure 5.5, Figure 5.6 and Figure 5.7, there is a trade-off between the training time instance usage fee. When we use an f1.2xlarge for the whole time, the

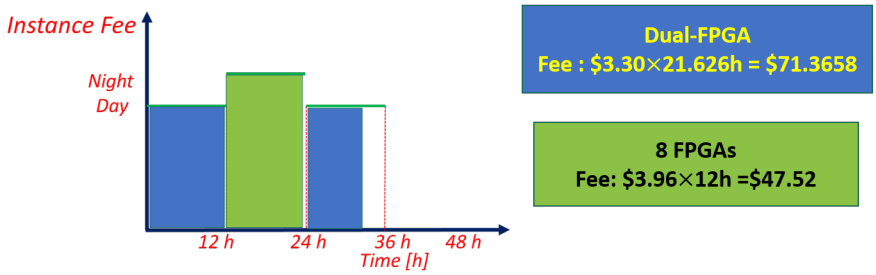

Figure 5.10: An example of optimizing the resource size between dual FPGA and 8 FPGAs

instance fee is very cheap, while the training time is huge. When we use an f1.4xlarge for the whole training, the training time is reduced while the instance usage fee is relatively high. When we use an f1.16xlarge, which is the biggest resource in AWS, for the whole training, the training time is reduced very much while the instance usage fee is very high. These results indicate that as we try to reduce the cloud usage fee, the training time definitely increase.
However, our method can optimize this trade-off. By changing the size of the instance during the training depending on the instance usage fee at that time, we can control the training time as well as the instance usage fee. Figure 5.8, Figure 5.9 and Figure 5.10 show examples of changing the instance depending on day time and night time. We use a small resource for the daytime, whose instance is expensive than that of the nighttime, and use a large resource for nighttime. By doing this optimization, we can reduce the training time as well as the cloud instance usage fee. The overhead time for changing the size of the instance is only a few minutes (AWS).

Figure 5.2 shows the results of training DanQ using our definitions. Basically, when cloud users use the same resource until their applications finish, the cloud instance usage fee or the execution time is very high. However, using our system, which can change the size of the resource during the execution, we can provide each user the most optimized execution time or the cloud instance usage fee. As shown in Table 5.2, although the fastest execution is using the largest resources, which takes 29.00 hours, we can reduce the amount of instance usage fee by more than half of it by only taking 4.63 extra hours, which can be a great optimization for some cloud users who can wait for 33.63 hours and want to save the instance usage fee.

5.6 Discussion

For the optimization we mentioned above, the model we assumed is not the only case we have an advantage for changing the size of resources during the training. Suppose

Table 5.2: The Result of Instance Usage Fee with the Training Time

	Training Time [h]	**Instance Usage Fee [$]**
All Single-FPGA	48.38	51.79
All Dual-FPGA	36.89	91.96
All 8 FPGAs	29.00	271.92
Day: Single-FPGA Night: Dual-FPGA	42.59	58.01
Day: Single-FPGA Night: 8 FPGAs	38.61	77.66
Day: Dual-FPGA Night: 8 FPGAs	33.63	118.89

a cloud user wants to accelerate a larger application that needs few months to execute. In that case, our optimization has more advantages of providing an appropriate training time as well as an appropriate cloud usage fee for the user.

5.7 Summary

In this chapter, we showed our optimization method using examples of changing the instance size during the training. As the results, comparing a case of using 8 FPGAs for all time and a case in which we optimized the number of FPGAs during the training with our model, we obtained the result that we can save the cloud usage fee for 56.28% by only taking 16.00% extra time.

Chapter 6

Conclusion and Future Works

6.1 Conclusion

6.1.1 FPGA Implementation

There is a big task for improving the training time for deep learning, especially in the field of genomics. There are huge datasets of DNA sequences to estimate the chromatin effect.
Therefore, it is necessary to accelerate the training time, and we proposed a method of using an FPGA. We focused on BiLSTM Layer and implemented it on AWS EC2 F1 Instances.
As a result, we could accelerate the DanQ model by using a single FPGA by 1.05x compared to our CPU implementation. Besides, our implementation on 8 FPGAs gets 2.87x faster than the dual FPGA implementation and 6.00x faster than the CPU implementation.

6.1.2 Cloud Optimization

There are so many cloud users who concern the trade-off between the cloud usage fee and the execution time. This is because the cloud usage fee always changes each time.
Therefore, changing the instance size during the training depending on the cloud usage fee at that time leads to a better result in terms of the training time and the cloud instance usage fee.
Comparing a case of using 8 FPGAs for all time and a case in which we optimized the number of FPGAs during the training with our model, we obtained the result that we can save the cloud usage fee for 56.28% by only taking 16.00% extra time. Therefore, we can optimize the training time as well as the instance usage fee depending on the user's needs.

6.2 Future work

We consider that we can use more FPGAs to accelerate the training time. In this paper, we only focused on the BiLSTM layer. However, we can implement the whole model by using at least 4 FPGAs. We will use a technique of model parallelism to implement the whole model and use a technique of data parallelism to accelerate the training time. Moreover, we will use the advantage of cloud service. We will use multiple FPGA instances to dynamically optimize the training time, depending on the user's needs. We will change the number of instances during the training by using MPI or OpenMP, and we will provide the users the most optimized instance fee for the model training. We also want to focus on scheduling methods which can provide us the most efficient size of the resources at each time by estimating the future usage fee of cloud services using deep learning.

Chapter 7

Acknowledgment

In two years of master research and writing the thesis, many people supported me and assisted my research and the thesis.

I first would like to thank my supervisor, Professor Yasushi Inoguchi, who helped me so much for the research approach and gave me a lot of good advice for experiments. His insightful feedback pushed me to sharpen my thinking and brought my research to a higher level.

In addition, I would like to acknowledge an assistant Professor, Ryuta Kawano, who have discussed my research very friendly. He gave me a lot of good advice that deepens my understanding of my research

I also would like to thank my laboratory members, who have taught me how to use an tool or an software in detail. They also encouraged me to concentrate on my studies and successfully I could finish my thesis.

Finally, I would like to thank my parents and friends who took care of my mental health during my hard time doing my research and thesis.

Research Achievements

Conference Proceedings

- 稲葉貴大，河野隆太，井口寧．”ゲノム向けCNNにおけるFPGA実装”. 電気・情報関係学会北陸支部連合大会（2020）.
- Takahiro Inaba, Ryuta Kawano, Yasushi Inoguchi. "Optimized-FDanQ: Implementation of Hybrid Network for Genome Data on Cloud Multi-FPGA and its Optimization under Given Costs". IEICE Transactions on Information and Systems (Parallel, Distributed, and Reconfigurable Computing, and Networking), 2021. (under review)

Award

- 2020年度電気・情報関係学会北陸支部連合大会　IEICE学生優秀論文発表賞

Bibliography

[1] M. Aledhari, M. Di Pierro, M. Hefeida, and F. Saeed. A deep learning-based data minimization algorithm for fast and secure transfer of big genomic datasets. IEEE Transactions on Big Data (Early Access), pages 1–13, Feb 2018.

[2] Y. Bengio, P. Simard, and P. Frasconi. Learning long-term dependencies with gradient descent is difficult. IEEE Transactions on Neural Networks, 5(2):157–166, 1994.

[3] Stuart Byma, J. Gregory Steffan, Hadi Bannazadeh, Alberto Leon-Garcia, and Paul Chow. Fpgas in the cloud: Booting virtualized hardware accelerators with openstack. 2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines, 2014.

[4] Andre Xian Ming Chang, Berin Martini, and Eugenio Culurciello. Recurrent neural networks hardware implementation on fpga. CoRR, 2016.

[5] Stanislaw Deniziak and Slawomir Bak. Scheduling of distributed applications in hhpcaas clouds for internet of things. 2020 23rd International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS), April 2020.

[6] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. Journal of Machine Learning Research, 12:2121–2159, jul 2011.

[7] Yihong Gao, Huadong Ma, Haitao Zhang, Xiangqi Kong, and Wangyang Wei. Concurrency optimized task scheduling for workflows in cloud. 2013 IEEE Sixth International Conference on Cloud Computing, April 2013.

[8] Tong Geng, Tianqi Wang, Ahmed Sanaullah, Chen Yang, Rui Xu, Rushi Patel, and Martin Herbordt. Fpdeep: Acceleration and load balancing of cnn training on fpga clusters. 2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines, 2018.

[9] Geoffrey Hinton, Nitish Srivastava, and Kevin Swersky. Neural networks for machine learning lecture 6a overview of mini-batch gradient descent. 2012.

[10] Geoffrey E. Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan R. Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. CoRR, jul 2012.

[11] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. Neural computation, 9(8):1735–1780, Nov 1997.

[12] Dongup Kwon, Suyeon Hur, Hamin Jang, Eriko Nurvitadhi, and Jangwoo Kim. Scalable multi-fpga acceleration for large rnns with full parallelism levels. 57th ACM/IEEE Design Automation Conference (DAC), 2020.

[13] Sifei Lu, Xiaorong Li, Long Wang, Henry Kasim, Henry Palit, Terence Hung, Erika Fille Tupas Legara, and Gary Lee. A dynamic hybrid resource provisioning approach for running large-scale computational applications on cloud spot and on-demand instances. 2013 International Conference on Parallel and Distributed Systems, December 2013.

[14] Daniel Quang and Xiaohui Xie. Danq: a hybrid convolutional and recurrent deep neural network for quantifying the function of dna sequences. Necleic Acids Research, 44(11), Apr 2016.

[15] DMahendra Samarawickrama Ranga Rodrigo and Ajith Pasqual. Hls approach in designing fpga-based custom coprocessor for image preprocessing. 2010 Fifth International Conference on Information and Automation for Sustainability, 2010.

[16] Junnan Shan, Mihai T. Lazarescu, Jordi Cortadella, Luciano Lavagno, and Mario R. Casu. Power-optimal mapping of cnn applications to cloud-based multi-fpga platforms. IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS, 2020.

[17] Jian Zhou and Olga G Troyanskaya. Predicting effects of nocoding variants with deep learning-based sequence model. Nat Method, (12):931–934, 2015.

Appendix A

Implementation Alveo U200

A.1 The Experimental Environment of Xilinx Alveo U200

We implemented DanQ on Xilinx Alveo U200 Data Accelerator Card (Alveo U200). This FPGA is used for a data center so that it has large capacity of on-chip memory. The experiment environment is shown in Table A.1.

Table A.1: Hardware Utilization of BiLSTM

	BRAM	**DSP**	**FF**	**LUT**
Utilization(%)	2892	1	1	2
Utilization SLR(%)	8676	3	3	6

The detail information of Alveo (U200) is shown in Table A.2.

Table A.2: Information about Xilinx Alveo U200

off-chip memory (DDR)	64GB
Bandwidth for off-chip memory	77GB/s
PCI Express	Gen 3×16
LookUp Table (LUT)	892,000
On-chip memory (BRAM, Register)	2,364KB
DSP slices	6,840
Maximum Power consumption	225W

The detail of PC server and the software that I used for implementing on Alveo U200 is shown in Table A.3.

Table A.3: Information about Experiment PC

OS	CentOS 7.7
Memory	64GB/s
CPU	AMD Ryzen 7 3700X
SDAccel	2019.1
Vivado HLS	2019.1

We mainly used SDAccel and Vivado HLS as a tool to do my experiment. SDAccel is a Integrated Development Environment (IDE) for the implementation for actual FPGA. Vivado HLS is a tool for generating Register Transfer Level (RTL) codes from C/C++ codes using High Level Synthesis (HLS).

A.1.1 Hardware Utilization

We first analyzed the hardware utilization for the BiLSTM layer by getting an HLS report from Hardware emulation in SDAccel environment, which checks the correctness of the logic generated for the compute units and tests the functionality of the logic that will be executed on the FPGA. The table A.4 shows the estimation of hardware utilization of BRAM, DSP Block, FF and LUT using Xilinx Alveo U200. Here, Utilization SLR represents a percentage of utilization in one SLR. Since there are 3 SLRs in Alveo U200, we can accept until 300% of utilization SLR by dividing the kernel into three kernels.

Table A.4: Hardware Utilization of BiLSTM layer

	BRAM	**DSP**	**FF**	**LUT**
Utilization(%)	244	9	11	18%
Utilization SLR(%)	732	28	33	55%

According to the table A.4, the utilization of BRAM is 244%, which is exceeding the capacity of BRAM in Alveo U200. Therefore, we need quantization to decrease the size of parameters such as weight, activation, gradient and error to make BRAM utilization smaller than the maximum utilization of Alveo U200.

Table A.5 shows the estimation of hardware utilization of other layers except for the BiLSTM Layer.

Table A.5: Hardware Utilization of all layers except BiLSTM

	BRAM	**DSP**	**FF**	**LUT**
Utilization(%)	2892	1	1	2%
Utilization SLR(%)	8676	3	3	6%

According to this table, BRAM utilization is 2892% and totally exceeding the maximum utilization of Alveo U200. The reason for this is that two full-connected layers at the end of DanQ model are consuming most of the BRAM utilization. This is because all units in the full-connected layer have connections to all the adjacent units, making the size of parameters much larger than other layers.

A.1.2 Execution Time

Table A.6 shows the comparison of the execution time between the implementation on Xilinx Alveo U200 and others.

Table A.6: Comparison of Execution Time for one Input for BiLSTM

	Forward (ms)	**Backward (ms)**	**Total (ms)**
CPU	441.635	755.445	1197.08
single-FPGA(Alveo U200)	207.239	310.478	517.717
single-FPGA(AWS)	388.165	750.068	1138.233

Appendix B

Comparison with GPU

We also implemented all layers of DanQ on P100 GPU using Pytorch as a deep learning framework. The results are shown in Table B.1.

Table B.1: Comparison of Execution Time for one Input for BiLSTM

	Forward (ms)	**Backward (ms)**	**Total (ms)**
GPU (P100)	7.324	9.434	16.758
CPU	441.635	755.445	1197.08
single-FPGA(AWS)	388.165	750.068	1138.233
8 FPGAs(AWS)	41.077	109.055	150.132

As the results, the execution time for using GPU outperforms others. It is very difficult to accelerate the whole DanQ using single FPGA compared to single GPU. This is not only because the GPU using the a deep learning framework are very much optimized, but also because the hardware utilization that is needed to implement DanQ on FPGA is large.