POLITECNICO DI TORINO

Master's Degree in Electronic Engineering

Master's Degree Thesis

Hardware Accelerator for LSTM Neural Networks using High-Level Synthesis

Supervisors

prof. Massimo PONCINO

doc. Daniele JAHIER PAGLIARI

Candidate

Chen XIE

IP Number: 250137

Academic Year 2019-2020

Summary

Neural networks are widely used in applications such as machine translation, speech recognition, etc. Among the different types of neural networks, recurrent neural networks (RNN) based on the Long Short-Term Memory (LSTM) architecture have become popular for elaborating time series.

To improve accuracy, the size of LSTM models continues to grow. Matrix-vector multiplications (MxV) are the most computation-intensive and time-consuming operations involved in LSTM inference. In order to perform these operations with high performance and low power consumption, Field-Programmable Gate Arrays (FPGAs) have become popular to accelerate LSTM inference. Based on FPGAs, finding the best accelerator architecture for a given objective and combining it with algorithm-level optimizations become the hot issues. In particular, the most common optimizations for LSTMs consists in using weight pruning to reduce the number of computations and memory occupation, transforming the dense MxV into a sparse matrix-vector multiplication (SpMxV). Accelerating SpMxV requires solving new issues, such as managing unstructured sparse matrices and their corresponding irregular memory access patterns.

In this thesis, a new LSTM accelerator for FPGAs is proposed, which addresses the two aforementioned problems. The design space exploration complexity is tackled using high-level synthesis (HLS), which allows the generation of a large number of different results starting from the same high-level specification changing some synthesis directives. Hundreds of accelerator implementations have been realized, among which a system designer could select depending on his/her requirements. On the another hand, the proposed accelerator is made compatible with a popular constrained pruning methods for LSTMs, known as Bank-Balanced Sparsity (BBS), which can maintain model accuracy at a high sparsity level while still enabling an efficient FPGA implementation.

The proposed design has been written in C++, synthesized using Xilinx Vivado HLS and flashed onto a Xilinx Zynq System-on-Chip (SoC). These SoCs include an ARM processor besides the FPGA, which has been programmed to trigger the accelerator and collect results by means of a software driver. Both design versions with DMA and without DMA are implemented. After implementation,

the performance of the accelerator for three different LSTM sizes are evaluated, with unoptimized and optimized cases respectively. The performance comparison between the implementation with DMA and the one without DMA are also discussed. **15x**, **30x** and **82x** speedups are achieved in unoptimized version, optimized version without DMA, and optimized version with DMA respectively, with low reasource occupation and power consumption.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Nowadays, artificial neural networks (ANNs) [1] have become one of the most popular research fields because of their higher prediction accuracy comparing traditional machine learning algorithms. In particular, in some latency-sensitive applications such as speech recognition [2] and machine translation [3], Recurrent Neural Networks(RNN), which are used to process sequence-related data, have been now considered the state-of-the-art. These deep learning models receive the output of the previous time step, and use it as a part of input, to perform the operations in current time step [2]. Because of this property, they are widely used in prediction of time series. However, traditional RNN networks have problems with long-term dependencies, which means that when they perform learning on long sequences of data, the gradient vanishing and gradient explosion issues appear [4]. Therefore, a variety of advanced algorithms are proposed.

Long Short-Term Memory (LSTM) networks are a kind of variant of RNN model, which can address the aforementioned problems [5]. Inside the LSTM network, there are special units called LSTM units, which consist of three gates: input gate, forget gate and output gate.

Despite the good performance of LSTMs, the increasing complexity of computation logic and the growing size of models lead to problems in power consumption, storage abilities and operation efficiency. These issues bring new challenges when these networks are used in low-power embedded systems.

Among the computation logic of LSTMs, Matrix-vector multiplications (MxV) are the most computation-intensive and time-consuming operations involved. To speed up these operations, besides the algorithm-level improvements, hardware acceleration also becomes a common choice. Nowadays, there are three popular platforms to support neutral networks: Graphics Processing Unit(GPU), Field-Programmable Gate Array(FPGA) and Application Specific Integrated Circuit(ASIC). With high performance, low power consumption and programmable flexibility, FPGAs [6] have become a great choice to accelerate LSTM inference in

power-sensitive applications.

However, accelerating LSTMs on FPGAs is not trivial, for two main reasons. First, finding the best accelerator architecture for a given objective (such as minimum latency or minimum energy) usually requires a time-consuming trial and error process, in which different architectures (in terms of amount of parallelism, data format, etc.) are iteratively designed and synthesized. Second, the hardware acceleration of LSTMs becomes tricky when combined with algorithm-level optimizations. In particular, one of the most common optimizations for LSTMs consists in using weight pruning to reduce the number of computations and the memory occupation, transforming the dense MxV into a sparse matrix-vector multiplication (SpMxV). Accelerating SpMxV requires solving new issues, such as managing unstructured sparse matrices and their corresponding irregular memory access patterns.

In literature, many research works have proposed dedicated hardware accelerators implemented on FPGA [7] [8]. However, few optimization of designs are proposed to support RNNs, in particular, LSTM networks. Some researchers such as YiWei Zhang [9] also proposed an accelerator based on FPGA for LSTM neural network prediction, but only focused on dense LSTM weight matrices. Therefore, storage resources become a bottleneck when facing large-scale neural networks.To solve this problem, traditional weight pruning techniques and Compressed Sparse Column/Compressed Sparse Column(CSC/CSR) formats [10] are provided to remove those parts of the weight matrices which have a negligible impact on the output accuracy, by setting a threshold, in order to achieve less storage resource space with acceptable accuracy. However, with the irregularity of sparse matrix-vector multiplications caused by the unstructured matrices, the optimization of performance and energy are limited on hardware accelerators [11]. To address these issues, various approaches are proposed. Shi et al [12] provided eSELL format for the sparse matrices, in order to solve the problem of area and data-access bandwith limitation. Kung et al [13] utilized Huffman-coded Nonzero Indication format to represent compressed matrices, which does not send row number but just a non-zero indication bit-stream and uses an on-chip counter to identify the locations of non-zero elements. In these studies, efficiency is increased but because of the complex encoding schemes introduced, the area of design is significantly larger.

On the another hand, further works suggest using balanced weight pruning to obtain structured sparse matrices, which can be more easily used to improve the performance of accelerator from a hardware perspective. The typical idea is coarser-grained weight pruning methods [14], which means splitting the weight matrices into blocks and performing the pruning inside each block. The main challenge of these techniques is the trade-off between accuracy and efficiency.

In this work, a sparsity pattern Bank-Balanced Sparsity(BBS) [15] is utilized, which further splits the matrix rows into equal-sized banks, and then performs the balanced pruning methods for each bank. Because of the finer splitting and

balanced pruning, the design can reach higher accuracy while using a decoding-free sparse matrix format, to further achieve high efficiency and save encoding-decoding area in the accelerator. Moreover, to explore the best architecture for different user requirements in specific application situations, the design space complexity is tackled using high-level synthesis (HLS). HLS allows to specify hardware designs using a high-level language (C/C++), and synthesize them to RTL. The most important is that it permits the generation of a large number of different results starting from the same high-level specification changing some synthesis directives.

Chapter 2

Background

2.1 Overview

Nowadays artificial neural networks(ANN) have become very popular in artificial intelligence field. Thanks to the growing abilities of data capturing and saving based on technique improvements, the researches of neural networks are progressing continuously and figuring out a lot of practical problems in a variety of fields such as medicine, automotive and economics etc.

Artificial neural networks are a deep learning model which is abstracted from the characteristics of biological neural networks. Definitionally, it is a kind of system which is consisting of many interconnected units, to process information through dynamic response to external inputs [16]. These process element units called neurons are connected with each other by different weights, which are used to multiply data passing through the connections. These neurons are split in different layers and process the information from previous layers and deliver the result to next layers. In general, a bias term can be added to the result. Based on this architecture, neural networks can deal with complex non-linear computations. Moreover, comparing to the traditional classifiers, the neural networks can be fed with raw data and automatically find the reasonable internal relationships and represent them suitably.

Because of these advantages, the researches on the artificial neural networks are continuously going, and gradually focusing on performing specific tasks, such as image restoration, speech recognition, etc. Based on the type of connection, we can identify two main types of artificial neural networks for classification and regression tasks: Feedforward Network, Recurrent Network. One particular type of feedforward network, Convolutional Neural Networks [17], have achieved excellent performance on image processing. Recurrent Neural Networks, in contrast, are more suited to time sequences processing.

In a variety of Recurrent Neural Networks, Jordan Network [18] and Elman [19] network are the earliest reccurrent neural networks facing sequential data. Their networks are called Simple Recurrent Network(SRN). After backpropagation(BP) algorithm [20] becoming popular, many works proposed novel approaches to train the recurrent neural networks under BP framework. Nevertheless, long-term dependencies problem [4] makes it hard to deal with long-time sequences, many optimizations have been appeared and one of the important breakthroughs is Long Short Term Memory (LSTM), which avoids the vanishing gradient problem [5].

Long short-term memory is a kind of time recurrent neural network which can deal with long-time interval events. Inside the model, LSTM units including input gates, forget gates and output gates, are utilized instead of neurons. In particular, input gate decides the inputs at the current time step and the updates of state from previous time step; forget gate decides the information to abandon; and the output gate decides the characteristics of the states propagated to the output. Further, LSTM augmented by "peephole connections" from the internal cells to multiplicative gates is proposed [21], which means that each gate can obtain the state information of the cells.

While the artificial neural network algorithms were flourishing, many studies started to move attention to the implementation of these models and corresponding accelerators on hardware. The hardware platform which can support neural networks includes Central Processing Unit(CPU), Graphics Processing Unit(GPU), Field-Programmable Gate Array(FPGA) and Application Specific Integrated Circuit(ASIC). The detailed comparison will be described in more detail in the following sections. Particularly, considering the low-power characteristic and programmable ability, FPGAs become a great choice to implement hardware accelerators.

With the appearance of high-level synthesis tools supporting FPGAs, the design time is greatly reduced. High-level synthesis(HLS) is a kind of techniques which produces hardware designs automatically by interpreting an algorithm-level description of a required behavior [1]. This technology can allow designer start at high-level language as C/C++ and corresponding register-transfer level(RTL) files are generated by the tools, further the digital hardware is implemented. HLS allows fast design space exploration. Many different accelerator implementations can be realized and the optimal architecture can be explored in different application situation, without struggling with complex HDL coding and expensive time costs.

The main architecture and corresponding techniques will be analyzed in more detail in the following sections.

2.2 Recurrent Neural Network

2.2.1 Basic RNN

Topology

Recurrent Neural Network(RNN) is a neural network with nodes connected in a chain [22]. As a kind of artificial neural network, the basic elements are neurons, the generic computational units performing a weighted sum of inputs. The general topological graph of RNN is shown as figure 2.1.

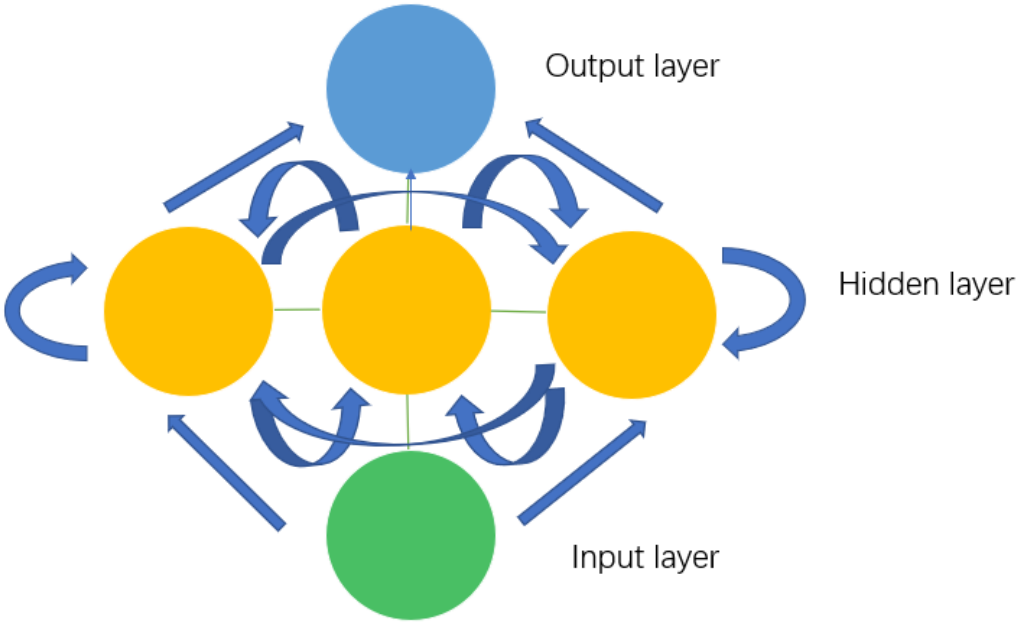


Figure 2.1: General Topological Graph of RNN

In this figure, we can see that there are three layers: input layer, hidden layer and output layer. In particular, input layer receives external inputs and send their output to hidden layer. Then output layer transforms the information receiving from hidden layer to desired scales and send it to outsides.

Moreover, in RNN, it is clear that the neurons in the hidden layer also have connections between each other, comparing to feed-forward neural networks. Because of these internal connections inside hidden layer, every time step neurons in the hidden layer can share data. In general, the core of RNN is described as a directed graph, which is a chain unrolled on time, from a networks with loops, as figure 2.2. Every time step, the information is copied from previous time step, and delivered to the same network in next time step. This permits the persistence the

persistence of information.

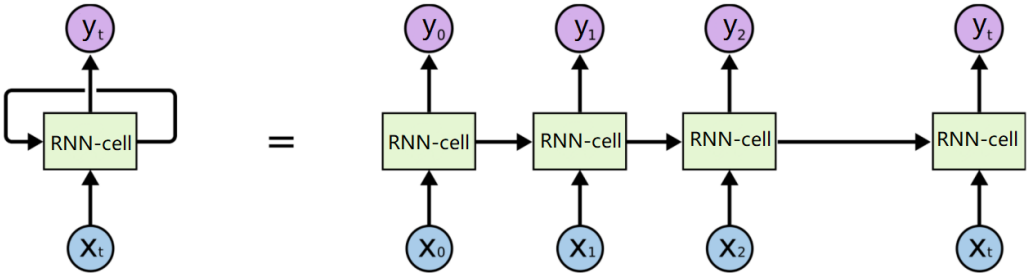


Figure 2.2: Unrolled RNN

Assume a sequence of values $X = \{x^{(1)}, \ldots, x^{\tau}\}$, where t is time step and τ is the unfold length of recurrent neural network. For each time step, the RNN-cell can be represented as [22]:

$$h^{(t)} = f\left(s^{(t-1)}, X^{(t)}, \theta\right) \tag{2.1}$$

where the h denotes the system status of recurrent neural networks, which describe the changes of all point with time step, in a given space. s is internal status which describes the summary of the important features of the previous inputs. θ denotes the weight parameter inside the cell. And f means the neural network used to perform the predictions.

In general, there is also output node, which can be written as:

$$o^{(t)} = vh^{(t)} + c \tag{2.2}$$

where v and c is weights parameters. Depending on the architecture of recurrent neural networks, the result of output points can be export outsides through an function. In other word:

$$\hat{y} = g(o) \tag{2.3}$$

Neuron

Neurons are responsible for the processing information. Each neuron receives information from the previous layer, and sends computing results to other connected neurons in next layer. The connection between two neurons has different weight, and these weights can be modified during training.

Assume one neuron shown as figure 2.3, the input from previous layer is $x0, x1, x2...xn$, and the weights in corresponding connections are $w0, w1, w2...wn$. With a bias b, the neuron can be shown as a linear model to perform a weighted sum.

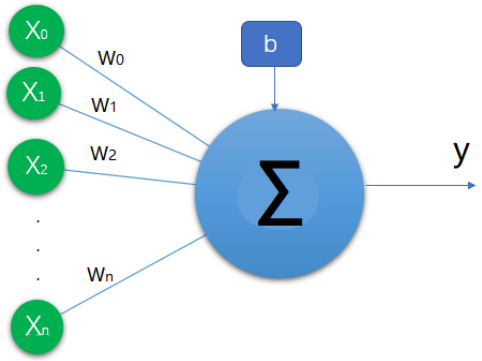


Figure 2.3: The neuron

However, to deal with non-linear problems, activation functions are introduced. Therefore, the formula of the neuron model is shown as:

$$y_i = h\left(\sum_{i=1}^{n} w_i x_i + b\right) \tag{2.4}$$

where h is activation function.

By introducing activation functions for each neuron, the linear sum of input becomes non-linear result and further the entire network has been transformed into a non-linear model. Typically, there are different activation functions can be chosen, and the most common ones are shown as following:

Activation Functions

- Sigmoid Function

$$h(x) = \frac{1}{1 + \exp(-x)} \tag{2.5}$$

Sigmoid function is one of the most popular activation functions. It can map the input into the range of (0,1) and always used on the output of hidden layer. The advantage is the smoothness and easiness to obtain derivation, but it is not easy to compute in hardware and generates vanishing gradient problems [23].

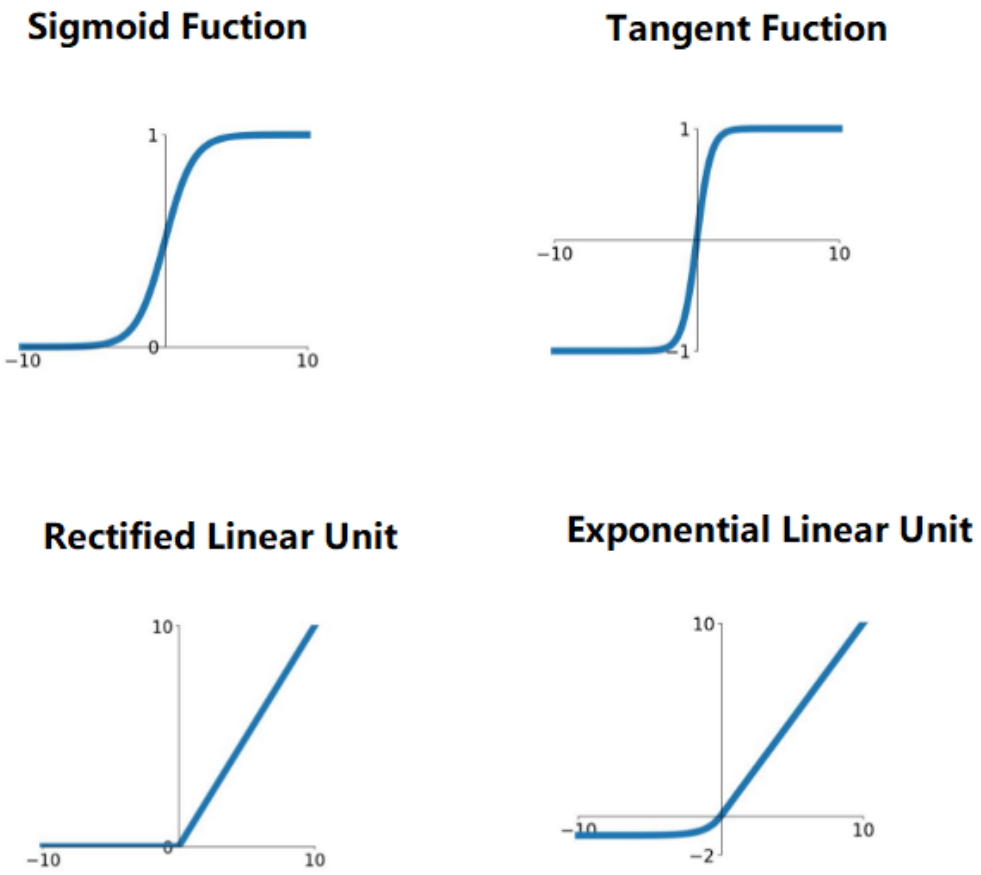


Figure 2.4: Activation Functions

- Tangent Function

$$h(x) = \tanh(x) = \frac{\exp(x) - \exp(-x)}{\exp(x) + \exp(-x)} \tag{2.6}$$

  Tangent function is also popular and has similar characteristics to sigmoid functions, also the same disadvantages. It is a symmetric function centered with zero. Therefore, it has better rate of convergence than sigmoid function.

- Rectified Linear Unit

$$h(x) = \max(0, x) = \begin{cases} 0 & x \leq 0 \\ x & x > 0 \end{cases} \tag{2.7}$$

Rectified Linear Unit(ReLU) is an activation which solves the gradient vanishing problems. In general, the most representative ones are ramp function and Leaky ReLU, whose equation is:

$$h(x) = \max(\alpha x, x) \quad \alpha \in (0,1) \tag{2.8}$$

ReLU allows for efficient gradient propagation in deep networks, is usually regarded as better activation function than previous logistic functions.

- Exponential Linear Unit

$$h(x) = \begin{cases} \alpha(\exp(x) - 1) & x \leq 0 \\ x & x > 0 \end{cases} \tag{2.9}$$

The exponential linear unit(ELU) is also proposed to solve gradient vanishing problems. This activation function merged sigmoid function and ReLU, the right part of curve is linear, which focuses on gradient problems; the characteristic of left part provides a noise-robust deactivation [23].

Although RNN has the ability to achieve predictions by previous information thanks to its recursive architecture, when related information appears with long time intervals, the vanishing gradient described previously will lead to long-term dependencies problem [4]. Many studies made the effort to propose improved algorithm and lots of novel architectures are introduced, including Neural History Compressor(NHC), Long Short-Term Memory(LSTM), Gated Recurrent Unit(GRU), and Independent RNN [24] [5] [25] [26].

2.2.2 Long Short-Term Memory

LSTM unit

As an important breakthrough, Long Short-Term Memory(LSTM) is proposed as a variant of RNN to solve gradient vanishing or exploding problems.

Inside this neural network, LSTM units are used to replace neurons in hidden layers. Each LSTM unit receives the input value of current time step $\mathbf{x}_t$ and the output value of past time step $\mathbf{h}_{t-1}$. The basic structure is shown as in Figure2.5.

We can see that in the LSTM unit, there four gates: input gate i, forget gate f, output gate o and cell input g, while the traditional neurons in recurrent neural networks jut have the last one. Moreover, there is a cell which can "remember" the information, and the updates will happen at each time step. These gates are responsible to choose what information to keep and what to abandon, and to change

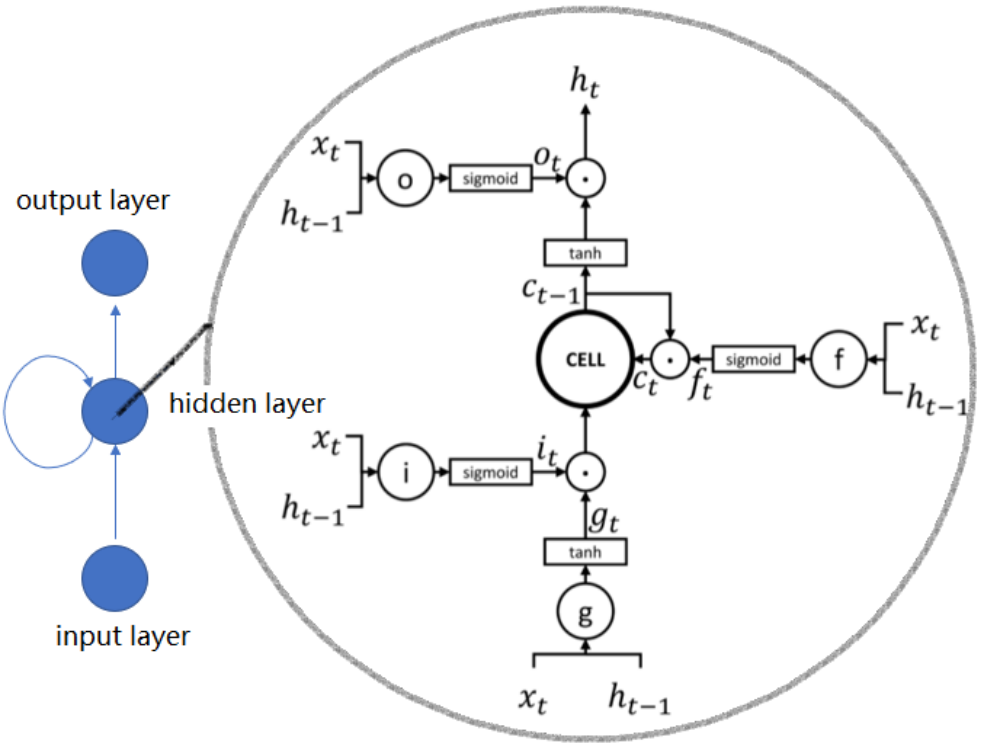


Figure 2.5: The Basic Structure of LSTM units

the cell state. After the gates, there are sigmoid functions and tanh function to deal with the output of gates.

The most important part in LSTM is forget gate, then input gate and output gate [27]. Sigmoid function will map the output from these gates into range(0,1). It works as a "switch", to decide if the information can affect cell state. If the result from input gate generates value closed to zero after sigmoid function, then it means the input value is blocked, otherwise new inputs information will be added to cell state; if the result from forget gates closed to zero after sigmoid function, the "memory" of the cell will be "forgotten". After updating current state, output gate need to decide the output results to send outsides at current time step.

LSTM Inference Process

In this project, the acceleration is focusing on the inference optimization. The LSTM neural network works as following equations:

$$i_t = \sigma\left(W_{xi}x_t + W_{hi}h_{t-1} + b_i\right) \tag{2.10}$$

$$f_t = \sigma\left(W_{xf}x_t + W_{hf}h_{t-1} + b_f\right) \tag{2.11}$$

$$o_t = \sigma\left(W_{xo}x_t + W_{ho}h_{t-1} + b_o\right) \tag{2.12}$$

$$g_t = \tanh\left(W_{xg}x_t + W_{hg}h_{t-1} + b_g\right) \tag{2.13}$$

$$c_t = f_t \odot c_{t-1} + i_t \odot g_t \tag{2.14}$$

$$h_t = \tanh\left(c_t\right) \odot o_t \tag{2.15}$$

where x_t denotes input value at current time step; h_{t-1} denotes output value of hidden layer at previous time step; W denotes corresponding weights in the connection and b denotes a bias to be added to the result of sum of weighted inputs; $\odot$ denotes element-wise multiplication; σ denotes sigmoid function. And i_t, f_t, o_t, g_t and c_t are the activated result of input gate, forget gate, output gate, cell input and cell activation respectively.

The working process is as following:

- **Step 1:** Forget gate is the first step of the neural network. x_t and h_{t-1} are fed into forget gate and the result goes into sigmoid function to control if the forgetting of the information, which is shown as equation 2.11.

- **Step 2:** The second step is to decide the information to store in cell state. There are two information sources: one is the value provided by input gate i_t; another one is the candidate values g_t. These two inputs pass sigmoid function and tanh function respectively to achieve nonlinearization , and then updates the cell state together. This process is shown as equation 2.10 and 2.13.

- **Step 3:** The third step is to updates cell states. First the previous cell state perform multiplication with value of forget gate, to "forget" some unimportant information. At the same time, the value of i_t and g_t also perform multiplication to obtain the updated content. Then these two result are added together to obtain current cell state, shown as equation 2.14.

- **Step 4:** Last step is to decide which part of cell state can be output. The cell state will multiply the value controlled by output gate after a tanh function. The output h_t then will be fed again at the next time step. This process is shown in equation 2.12 and 2.15.

Main Computational Operations

According to the formula of LSTM models described before, we can see that in LSTM inference, main computational Operations are matrix-vector computation, element-wise computation, and activation functions. The whole computation model can be shown as Figure2.6 [9].

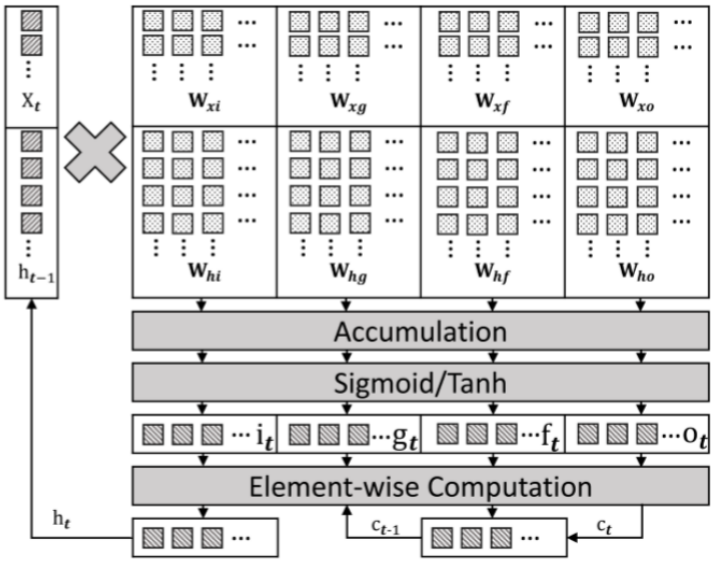


Figure 2.6: The Whole Computation model of LSTM

In particular, matrix-vector multiplications (MxV), as the most time-consuming operations, consisting of numerous dot product operations. These dot product operations are performed on each weight matrix row and the input vector composed by the concatenation of x_t and h_{t-1}. Assume size of input vector x_t is **INPUT_SIZE** (in fact the size of input layer), size of h_{t-1} is **HIDDEN_SIZE** (the size of hidden layer), then the size of this dense input vector is **(INPUT_SIZE+HIDDEN_SIZE)**, which must be exactly equal to the number of rows in matrix. Because of the weights distributed on the connections of four gates, the number of matrix column is equal to **(4*HIDDEN_SIZE)**. Therefore, number of dot products will be **(INPUT_SIZE+HIDDEN_SIZE)*(4*HIDDEN_SIZE)**. It is clear that as the size of matrix growing, the computation becomes more time-consuming. Each of these dot products involves **INPUT_SIZE + HIDDEN_SIZE** multiplications and additions.

Weight Pruning

To ensure the accuracy of the neural network models, the sizes of networks are constantly growing, which challenges the computation costs and resource storage. In particular, with three more gates in LSTM units and even peephole connections, the LSTM network has a huge scale of parameters, especially weight matrices. However, not all of the connections provide same contribution to the final result and there exists a lot of redundancy. Therefore, many compression techniques were proposed.

- **Deep Compression**

  Deep Compression [28] is a threshold-based weight pruning technique. It prunes away the absolute values of weights bellow the threshold, and then the remaining weights take part in retraining. After pruning, the dense weight matrices converts to sparse matrices, which needs less memories and computation costs. However, because the pruning faces to the whole weight matrices, it leads to unstructured sparse matrices and further limits the efficiency achievable on hardware accelerators.

- **Coarse-grained Pruning**

  To avoid the irregularity of computation and memory accesses, coarse-grained pruning method [11] is proposed, which means restricted pruning. The main idea of this method is to perform pruning on matrix blocks, instead of individual elements. The maximum magnitude or the average magnitude of elements in one block is used to compare with pre-defined threshold, and then the entire block is pruned or kept depending on the comparison result. The final sparse matrices become regular but at the same time, the accuracy of the model is reduced. It happens that some important connections are pruned when they hides in a block with small average magnitude.

2.2.3 Field-Programmable Gate Array

Because of the feature of neural network architecture, there is a piratical significance to sped up the computation, particularly in latency-sensitive application. Hardware accelerators, which is a specific hardware circuit to perform a part of computation of neural networks, becomes a popular choice. Comparing with performing the same algorithm on a general purpose processor, specific hardware circuits consume less power with high performance. Nowadays, the common choices are Application Specific Integrated Circuit(ASIC), Graphics Processing Unit(GPU), and Field-Programmable Gate Array(FPGA).

- **ASIC**

ASIC is a specific circuit aiming at certain requirements and applications. Both computation ability and efficiency of this hardware are customized based on desired algorithms. Therefore, ASIC has advantages in the aspect of power comsuption, circuit size, computation performance and efficiency. On the other hand, customized circuits also have the drawbacks in flexibility to implemented different algorithm. At the same time, high cost of development and long design period also become limits.

- **GPU**

  GPU is a computation unit which is good at performing parallel computation. With large number of computation units and high-speed memory, GPU preform great at graphics processing. The abilities to deal with coarse-grain parallelism makes GPU find its place to be a hardware platform for neural networks, since a great number of parallel executions exists in neural network algorithms. Comparing with ASIC and FPGA, GPU consume more power and has lower efficiency.

- **FPGA**

  FPGA is an integrated circuit with semi-custom attributes. It is designed to be configured by a customer to implement his/her design and achieve related requirements. Because of "field-prgrammble" feature, FPGA has the flexibility to realize different algorithms with shorter design period. However, the possibility of customizing the hardware configuration to the target applications makes them more efficient than GPUs. Hence with high performance and low power consumption, FPGA becomes a great choice to implement neural network models and explore the best architecture.

In this project, considering the performance and development period, we select FPGA as the hardware platform to design the accelerator.

FPGA is a collection of a large number of Configurable Logic Blocks(CLBs) or Logic Elements(LEs), and these basic elements are organized as a matrix array of rows and columns [29]. One of these basic elements is shown as figure 2.7.

As basic building blocks, LEs are composed of Look-Up Tables (LUTs) coupled with flip-flops, and a multiplexer is used to select the combinational or sequential output. CLBs and LEs are exactly responsible for hardware programming. With the matrix comprising by these elements, FPGA also has programmable interconnections which allow to route signals from one CLB or LB to another. And configurable I/O blocks allow to connect to the external world. One of modern Xilinx FPGA architecture is shown as Figure 2.8. We can see that besides CLBs/LEs, modern FPGAs also have embedded memories (BRAMS) and embedded DSP blocks for arithmetic operations.

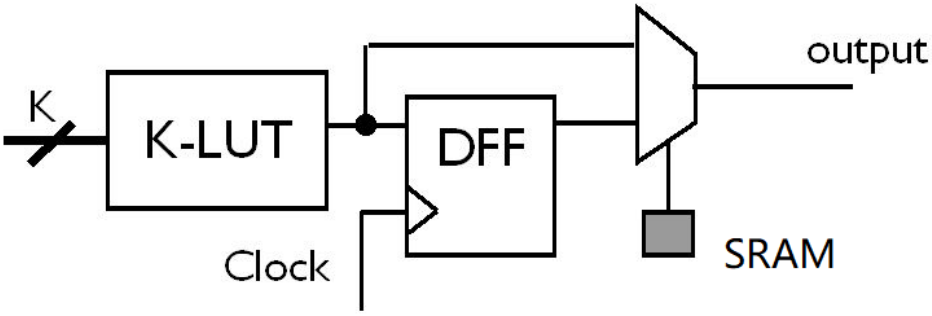


Figure 2.7: The Basic Structure of Logic Elements

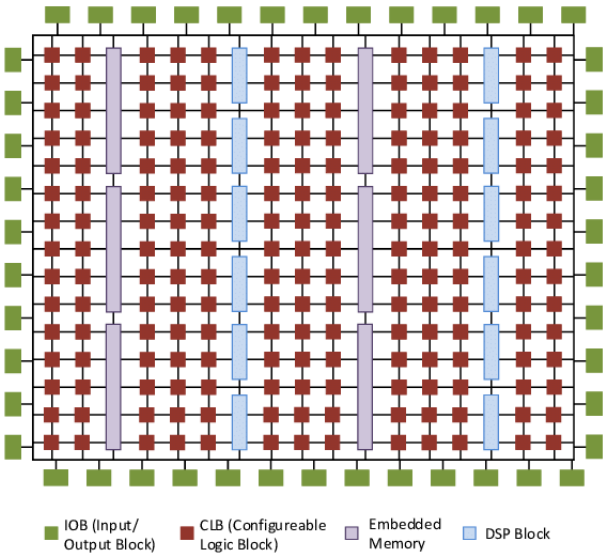


Figure 2.8: Modern Xilinx FPGA Architecture

Because multiply computation logic architectures exists, FPGA is suitable for parallel computation to perform independent operation in neural network algorithms. Moreover, it is convenient to optimize the accelerator architectures by pipeline to improve throughput.

2.2.4 High-Level Synthesis

Introduction

High-level synthesis(HLS), also called C synthesis, electronic system-level(ESL) synthesis, algorithmic synthesis or behavioral synthesis, is a technique to transform a C specification into a register transfer level (RTL) implementation that you can synthesize into a digital hardware [30]. Early HLS explored a variety of input specification languages [31], and nowadays C language family including C/C++, SystemC, ANSI C and also MATLAB is used in general. After analyzing, constraining and scheduling, the high-level language which describe the hardware behaviors can transform into corresponding hardware description language(HDL) such as verilog and VHDL, and then the design is synthesized by logic synthesis tools. After synthesis, co-simulation between C testbench and RTL implementations is performed, in order to perform verification of the RTL.

High-level synthesis powered up FPGA in the competition of hardware accelerators to support neural networks, show as figure 2.9 [32].

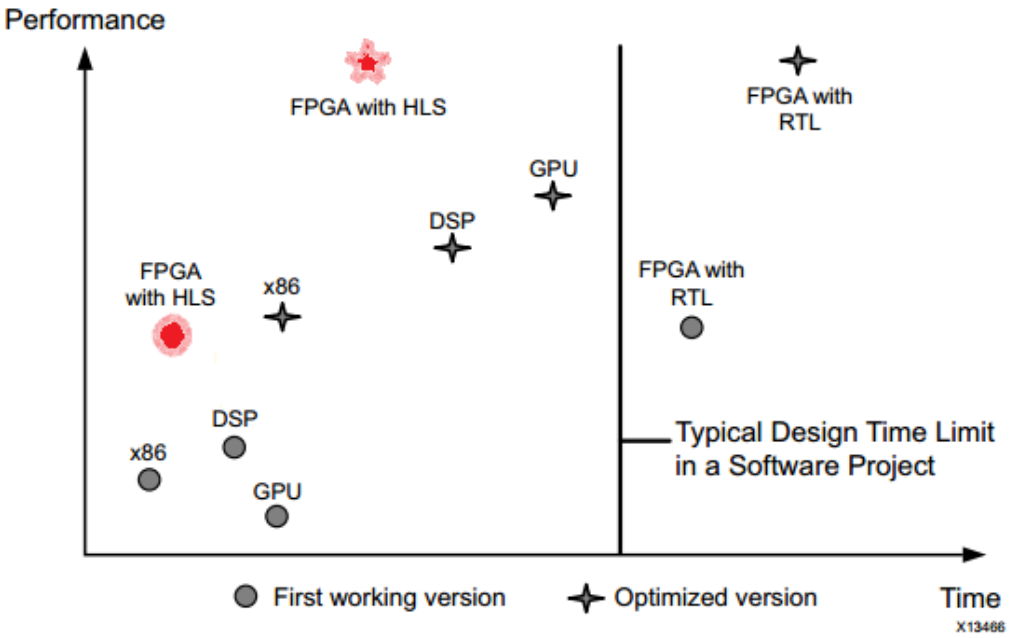


Figure 2.9: Design Time vs. Application Performance with RTL and Vivado HLS compiler

Comparing other hardware platform such as GPU, FPGA with RTL has high performance but need more development time. However when applying HLS, as

red point and star shows in the figure, development time is decreased sharply, even lower than DSP and GPU, while performance keeping at high level. This makes users paying more attention on behavior design without caring about specific implementation at low level, therefore productivity for hardware designers is improved.

Vivado High-Level Synthesis

Vivado Design Suite [32] is a powerful software suite produced by Xilinx. The Vivado HLS compiler enables C specification language to be directly targeted into Xilinx devices. It includes multiple components to support whole design process: Vivado Simulator, Vivado IP Integrator and Vivado TCL Store.

- **Vivado Simulator**

  Vivado Simulator is a compiled-language simulator, to perform C verification and RTL verification. TCL scripts, mixed language and IP are supported.

- **Vidado IP Integrator**

  Vivado IP Integrator can perform integration of Intellectual Properties (IPs) designed by user using Vivado High-Level Synthesis tools, and IP from Xilinx IP libary.

- **Vivado TCL Store**

  Vivado TCL Store is a system to support scripting. This system is based on Tool Command Language (TCL) stands and TCL scripts can be operated to invoke Vivado functions and makes large number of synthesis and simulations achievable.

HLS Design Flow

The general HLS design flow is shown as Figure 2.10 [33].

Designers write the behaviors of desired design and testbench in C specification languages, then perform C simulations to validate the C source code. After successful simulations, the synthesis is performed to create an initial RTL implementations written by verilog and VHDL. And then the RTL implementation could be transformed into both low-area and high-throughput implementation by adding different optimization directives, without changing the C source code. When synthesis is finished, the co-simulation could be performed, combining C testbench and RTL implementation, to achieve RTL verification. RTL designs generated by high-level synthesis are then packaged as IP and exported. In Vivado Design Suite environment, previous IP is added into IP Catalog and finally used in a SoC design.

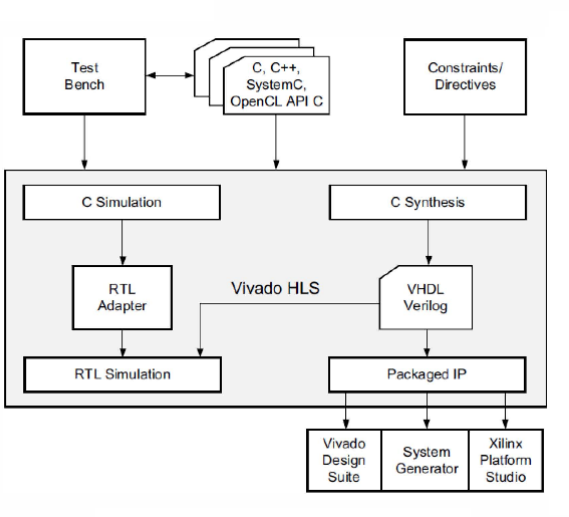


Figure 2.10: Vivado HLS Design Flow

HLS Phases and Mapping

In particular, high-level synthesis phases [32] are introduced here. There are three main phases including:

- **Scheduling**

  Scheduling is ordering the operations at each clock cycle depending on clock cycle length, operation time and optimization directives. When clock period is shorter than execution time of operations, high-level synthesis automatically schedules the operations more than one clock cycles.

- **Binding**

  Binding is to decide the hardware resource that will implement each scheduled operation. This phase will consider the specifications of target device.

- **Control logic extraction**

  The control logic is extracted automatically to generate corresponding finite state machine(FSM), which manage the operations in RTL.

The mapping relationship between C code and RTL implementation is as following:

C Code	RTL Implementation
Top-level function arguments	RTL I/O ports
C functions	Blocks in the RTL hierarchy
Loops	By default the logic for one iteration of the loop
Arrays	Block RAM or UltraRAM

Table 2.1: The Mapping Relations between C Code and RTL

Chapter 3

Related Works

With the Booming of deep neural networks, hardware accelerators to support corresponding algorithms in different application situations become hot issues. Because of the feature of neural network models, the huge scale of parameters and numerous element-wise computations challenge the resource storage and computation costs. Even more, the trade-off between efficiency and area consumption when application in embedded environments. To achieve fast response with limited resources, more solutions for neural network models have been proposed, both in hardware level and software level.

In software or algorithm level, the popular solutions are mainly focusing on approximate computing paradigm. There are two categories exist recently:

• change the floating point operations into fixed point ones, in order to reduce corresponding size of operands and operators. The most common data type in neural networks is 32-bit. When implementing in embedded systems, less bitwides such as 16-bit even 8-bit may be a better choice, even loss a little accuracy that will not affect much on final result of models.

• compression techniques, including weight pruning methods and compression representation format. The main idea of this technology is to remove part of less important connections inside the neural networks, to make dense weight matrices converting to sparse ones. The number of element-wise computations such as element- wise multiplication between weights and vectors will be reduced. At the same time, irregular memory accesses leading by unstructured sparse matrices becomes new issues. Moreover, with some compression pattern format, encoding and decoding schemes may bring new challenges on resource occupations and computation costs.

In hardware level, the solutions fall on hardware platform selections and optimizations for the model implementations. GPU, ASIC and FPGA becomes hot choices for different application purpose. And when optimizing the hardware architecture, the parallel computation and pipeline techniques are often considered.

3.1 Hardware Implementation and Optimization

Many studies proposed interesting hardware design or optimizing solutions. Wang, Chao, et al. [34] proposed a scalable deep learning accelerator unit to achieve less latency at low-power level. S. Li et al. [35] provided an implementation for standard RNN. The main idea of the approach is that unfold the RNN models into a fixed number of time steps and achieve parallel computations.

D. J. Pagliari et al. [36] proposed an hardware accelerator at high level. Inside the hardware accelerator, microarchitectures are used to exploit the intrinsic parallelism. Moreover based on high level design, explicit parallelism and parallel subblocks granularity are discussed.

Facing the implementation of LSTM based on FPGA, Chang, A.X.M. and Culurciello, E.[37] provided three architectures: DeepStream, DeepStore, and DeepRnn, for different applications. These architecture use high DMA bandwidth or high resource occupation or trade-off between them, users can select what they want based on their requirements.

Zhang et al. [9] implemented an accelerator based on FPGA for dense LSTM network. Inside the matrix-vector computations, they provided Tiled Multiplication, overlap methods of computation and data access, to improve the throughput.

Guan, Yijin, et al. [38] ont only utilized tiled computations also, but performed communication optimization. When the parameters and inputs of neural networks with large scales, it is impractical to store all of them into FPGA which has a limited resources. Therefore, one of solution is consuming parameters and inputs during run-time inference. In their work, they implemented two input buffer groups and two output buffer groups to work in a ping-pong manner during data accessing, and a data dispatcher is also designed.

3.2 Software Optimization

In software/algorithm level, many researchers provided multiply solutions to reduce the costs of computational operations and widen the space to explore parallelism further for hardware implementations.

In the project finished by Ferreira Joao Canas and Jose Fonseca [39], they used 18-bit fixed point to make full use o the DSP48E1 slices available in the FPGA. And they also used polynomial Approximations to make activation functions transforms into the logic without restriction on resources and speed. To minimize the error brought by this approximation, Least Maximum Approximation strategy is used to find optimal polynomial.

Focusing on the compression techniques, various weight pruning methods and corresponding sparse formats are proposed. One of the common sparse format

is compressed sparse row/Column(CSR/CSC) format [40]. This sparse format based on deep compression, which pruned elements whose absolute value bellow threshold in the matrix. However, this compressed sparse format will bring two types of overheads: rearranging the non-zero elements and fetching vector elements to calculating address when decoding. Therefore, many novel encoding formats are proposed.

Shi et al [12] provided a strategy to rearrange the formats by eSELL format. This format is created to solve the problem of area and data-access bandwidth limitation for on-chip applications. In CSR/CSC, because of the non-zero values in weight matrices are put together and row/column location numbers are stored. It actually has no random-access problem in weight matrices, but the result has random access problem (output for next parts). To solve this problem,this paper tries to rearrange the non-zero values to obtain continuous data by using a single-port SRAM with wider port width. To further compress the sparse representation, it then encodes the column indices of each row to a 3-bit value.

Kung et al. [13] provided higher compression ratio algorithm on wider range of sparsity levels. Compare to CSC, it does not send row location numbers, but sends non-zero indication bit-stream and use on-chip counter to identify the locations of non-zero elements at run time. Then the stream is encoded with Huffman coding.

However encoding strategy also brings the overhead because of encoder and decoder architectures are required. Focusing on this, Cao, Shijie, et al. [15] proposed a decoding-free sparse pattern Compressed Sparse Banks(CSB), with related Bank-Balanced Pruning algorithm. In this work, the sparsity of weight matrices becomes bank-balanced, which means solving the irregular memory access problems. At the same time, comparing to anther coarse-grained pruning methods such as Block Pruning, this approach achieved high sparsity with higher model accuracy.

On higher-level optimizations for RNN, there are also some related works. One point is on deciding to execute RNN on the embedded system or on the cloud depending on input length. Based on this idea, an optimal input-dependent edge-cloud partitioning for RNN inference [41] is proposed to achieve processing short input sequences locally while offload long ones to the cloud. Another strategy is that to decide the number of RNN invokations depending on input complexity. A method called Dynamic Beam Width Tuning [42] is proposed. In this work, the Beam Width (BW) is modulated based on the current input, in order to improve the energy efficiency of encoder-decoder RNNs. Moreover, Dynamic Beam Search [43] method is also presented by operating BW to support sequence-to sequence neural networks inference on embedded systems, in order to reduce the inference time and energy.

Chapter 4

Hardware Accelerator

4.1 Motivation

Due to the computational complexity of neural networks, applying neural networks in latency-sensitive and resource-limited embedded environment becomes a new challenge. In this situation, not only neural network model accuracy needs to keep at high level, but also fast responses and low power consumption are required. Specific hardware accelerators to support corresponding neural network algorithms, become good choice to achieve this task in the perspective of hardware level.

Part of computations, in particular, the mulitply-and accumulate (MAC) operations, will optimized by the hardware accelerators which can perform these calculations in parallel way based on the hardware platform such as GPU, ASIC and FPGA. The main space to explore on these hardware accelerator is exactly at the optimization of computation costs and the parallelism in the processing logic.

In particular, for LSTM RNN models, to keep the models accuracy, the size of models continues to grow. Matrix-vector multiplications (MxV) are the most computation-intensive and time-consuming operations, as described in previous sections. Therefore, compressing of the weight matrices is necessary by various pruning methods. Considering the implementation of hardware, these pruning methods also need corresponding sparse formats to encode the location of non-zero values in sparse matrices.

Moreover, to explore the parallelism of processing logic, the best accelerator architecture for a given objective requires a time-consuming trial and error process, in which different architectures are iteratively designed and synthesized. High-Level Synthesis based on FPGA makes the trial achievable. With high performance and low energy consumption, FPGA is ideal hardware platform for embedded devices, and HLS makes the design period reduced sharply.

4.2 Objective

In this thesis, a new LSTM accelerator using high-level synthesis for FPGAs is proposed, which achieves algorithm-level optimizations and hardware-level optimizations.

For algorithm-level optimizations, in order to make the design compatible with weight pruning, a subset of these methods has been considered, known as constrained pruning, where the sparsity of matrices is not completely random. Specifically, the proposed accelerator is made compatible with a popular constrained pruning methods for LSTMs, known as Bank-Balanced Sparsity (BBS)[15]. In BBS, matrix rows are split into multiple equal-sized banks, and each bank keeps the same number of non-zero values (i.e. the same sparsity).

For hardware optimizations, thanks to high-level synthesis(HLS), the design space exploration becomes achievable without long time consumption. The design is written by high-level language and then synthesized by HLS tools into RTL description. With different synthesis optimization directives such as pipeline and unroll, a large number of different implementations can be generated without changing original high-level design architecture. Multiply implementations supporting different practical applications are generated for users. Then IP core is exported to integrated with IP blocks. After IP insertion and connection with processor, the bitstream files are generated and download into FPGA to achieve the programming of the FPGA. A software driver is written and run on the target devices system.

The proposed design has been written in C++, synthesized using Xilinx Vivado HLS and flashed onto a Xilinx Zynq System-on-Chip (SoC). These SoCs include an ARM processor besides the FPGA, which has been programmed to trigger the accelerator and collect results by means of a software driver.

4.3 Bank-Balanced Pruning

Pruning Algorithm

In this thesis, a sparse LSTM hardware accelerator is described, using the encoding-free BBS format for weight matrices.

As a kind of recurrent neural network, LSTM receives inputs x_t at current time step, and also outputs of the same layer h_{t-1} at previous time step. Considering the independence of calculations, x_t and h_{t-1} can merge into a long vector named input vector. At the same time, x_t and h_{t-1} inputs will have four weight matrices matching input gate i, forget gate f, output gates o, cell input g respectively, for a total of 8 weight matrices: W_{xi}, W_{xf}, W_{xo}, W_{xg}, W_{hi}, W_{hf}, W_{ho}, W_{hg}. This weight matrices can be merged together[9] with fixed location mapping x_t and h_{t-1}.

The bank-balanced pruning method is performed on the merged weight matrix.

Each matrix row is split into equal-sized sub-rows(banks), and based on pre-defined sparsity, the values in each blanks are sorted and then the weights with smaller absolute value are removed. For example, when the sparsity is 50%, and the bank size is 4, 2 elements with smallest absolute values will be removed, as figure 4.1

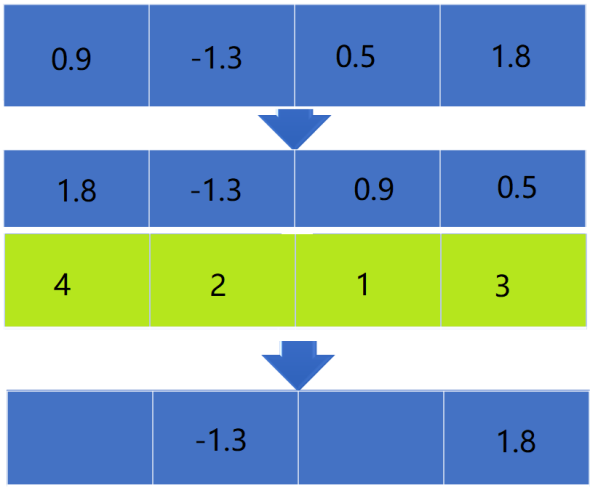


Figure 4.1: Bank Pruning

In this figure, the blue banks are the sub-rows split from dense matrix rows, and the green one is the original locations of elements in each bank before sorting. At first, the elements in the bank are sorted by their absolute value, and their original locations are recorded, shown as the first arrow. Then the two smallest(assume sparsity is 50%) values are removed, and the remaining elements are put back into the bank based on their original location information.

The dense weight matrices are pre-processed based on this pruning method by MATLAB, the algorithm is shown as following:

Compressed Sparse Bank Format

Compatible with Bank-Balanced Pruning algorithm, a sparse matrix format Compressed Sparse Bank(CSB) is used. Thanks to the balanced property of BBS, the need for decoding is eliminated, so that the overheads brought by decoding when performing sparse matrix multiplications are avoided. The representation of CSB is shown as Figure4.2.

In this figure, (a) is original pruned sparse matrix with 8 columns and 2 rows. The numbers shown on the top and left of the matrix are column numbers and row numbers respectively. In this matrix example, each row is split into 2 4-size banks with sparsity equal to 50%, i.e. the non-zero values in each bank is equal to 2.

```
for i=1:1:m                         %for each row
    for z=1:(n/BankNum):n           %for total bank
        for j=1:1:(n/BankNum)       %get the original bank
        bank(j)=A(i,j+z-1);
        end
        [~, index]=sort(bank,'descend');  %sort the bank

        for k=1:1:(n/BankNum)               %pruning
            if(k<=(n/BankNum*sparsity))
            pbank(index(k))=bank(index(k));   %obtain the element remaining in the bank
            end
        end
        for count=1:1:(n/BankNum)
        B(i,count+z-1)=pbank(count);          %write the pruned bank into pruned matrix
        end
        pbank=zeros(1,16);  %reset pbank
        bank=zeros(1,16);
    end
end
```

	0	1	2	3	4	5	6	7
0	A	B			C			D
1		E	F		G		H	

(a) Original Sparse Matrix

CSB								
VALUES	A	C	B	D	E	G	F	H
INDEX	0	0	1	3	1	0	2	2

(b) CSB Representation

Figure 4.2: CSB Representation

The CSB format includes two arrays named VALUES and INDEX respectively, as (b). VALUES array is responsible for storing non-zero values in the sparse matrix, and INDEX array is used to store bank internal indices of corresponding non-zero values. As the example in figure 4.2, the first non-zero element from the first bank A is put in the first location of VALUES array, at the same time, the A is the 0th element in this bank, therefore, 0 is put in corresponding location of INDEX array; then the first non-zero element from the second bank C is put in the second location of VALUES array, and its bank internal index is also 0, so that 0 is put in second location of INDEX array, and so on. When one row is rearranged, next row is following the same strategy.

This data rearrangement is to achieve inter-bank parallelism to make every successive elements can be directly fetched and fed for computation in parallel.

The CSB format generation is also pre-processed before hardware acceleration. The algorithm written by MATLAB is as follows:

```
for i=1:1:m                     %for each row
    bcount=1;
    for k=1:(n/BankNum):n       %for total bank
        for j=1:1:(n/BankNum)   %get the one bank
            bank(j)=pmatrix(i,j+k-1);
        end
        bidx=find(bank);  %return the indecies of non-zero values
        s=size(bidx);     %the #of non-zero value in each bank
        count=1;

%fill value array and index array
        for z=1:BankNum:(BankNum*s(2))
                values(z+(i-1)*(BankNum*(n/BankNum)*sparsity)+bcount-1)=bank(bidx(count));
                index(z+(i-1)*(BankNum*(n/BankNum)*sparsity)+bcount-1)=(bidx(count)-1);
                %note:-1 to start counting in 0
                count=count+1;
        end
        bcount=bcount+1;   %count #of banks
    end
end
```

4.4 Architecture

The main calculations of LSTM neural network are Matrix-Vector Computation, Element-wise Computation and the computation through Activation Functions. Therefore the overall architecture is mainly consisting with a Sparse Matrix-Vector Multiplication(SpMxV) Unit, Element-wise Vector Operation(EWOP) Unit, on-chip parameter matrices, i.e., Matrix Memory and Vector Memory which storing sparse weight matrix in CSB format and input vector(x_t and h_{t-1}) respectively. Moreover, a small controller is used to send load/store instructions. The overall architecture are shown as Figure 4.3.

Before sending data to the accelerator, the bank-balanced pruning method is performed and the sparse matrix is represented by CSB format in pre-processing. When receiving **LOAD WEIGHTS** instructions, the controller will send control signals to load the CSB values and indices from host processor into Matrix Memory, and bias and input vector into Vector Memory with **LOAD BIAS** and **LOAD VECTOR** instructions respectively. After loading, the controller will send control signals when receiving instruction **READ PARAMETERS**, to read CSB weight values, their indices and input vectors from Matrix Memory and Vector Memory respectively, and start the computations by controlling corresponding modules to execute.

Memories

There are two memories Matrix Memory and Vector Memory, are used to store CSB format weights and indices, and input vectors taking part in matrix-vector computation. Moreover, the bias vector is also stored in Vector Memory, to participate in the last step of SpMxV unit.

Therefore, inside Matrix Memory, there are two equal-size arrays: VALUES

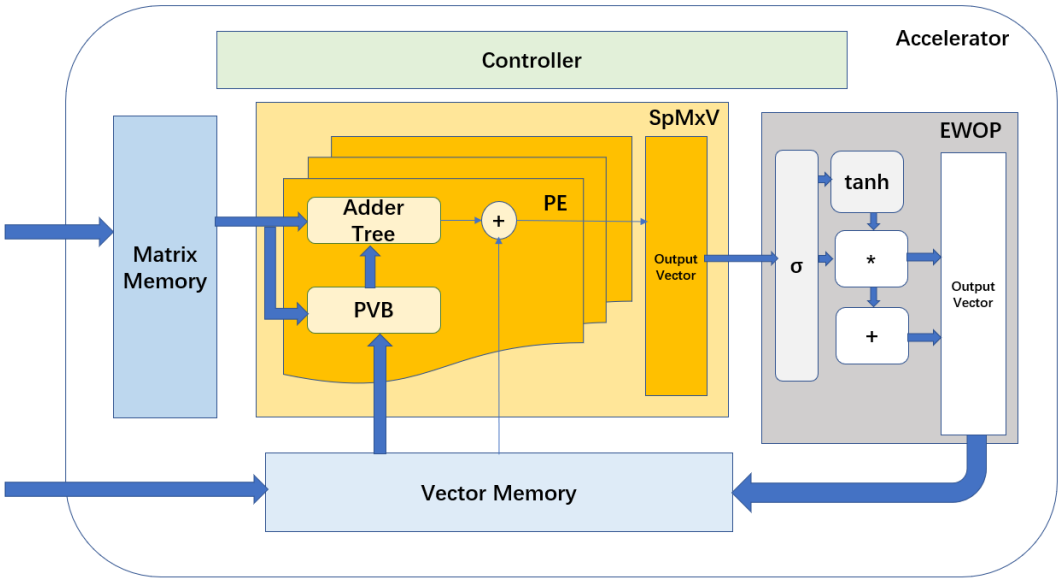


Figure 4.3: Overall Architecture of LSTM accelerator

array and INDEX array, as described in CSB format. The size of these two array is exactly the number of non-zero elements in whole sparse matrix. And also two arrays exist inside Vector Memory and named VECTOR array and BIAS array. The size of VECTOR array is equal to the number of matrix rows and of BIAS array is equal to the number of matrix columns.

Moreover, Vector Memory also receives the output from EWOP unit, and merge it into Vector array to generate new vectors for the next operation.

SpMxV Unit

The SpMxV unit is the unit computing the dot product of weights and vector. It consisting with multiple Processing Elements(PEs) which performs the multiplication between one weight row and the input vector. These processing unit can operate in parallel, and return a result element added with bias. These result will form a output vector fed to EWOP unit.

In particular, thanks to CSB format, the weights are already arranged by the order of fetching from each bank contiguously, so that the weights from each bank can be obtained concurrently. It benefits inter-bank parallelism. However, accessing the corresponding vector element, it will lead to random memory access, which block the inter-bank parallelism because BRAMs in FPGA do not support multiple R/W ports. Therefore the dense vector also needs to split into equal-size banks according to the bank partition of weights and indices.

Hence, when performing the dot-product operations, the indices recording the internal bank locations of weights are used to search the corresponding vector elements, as shown in Figure 4.4 and Figure 4.5.

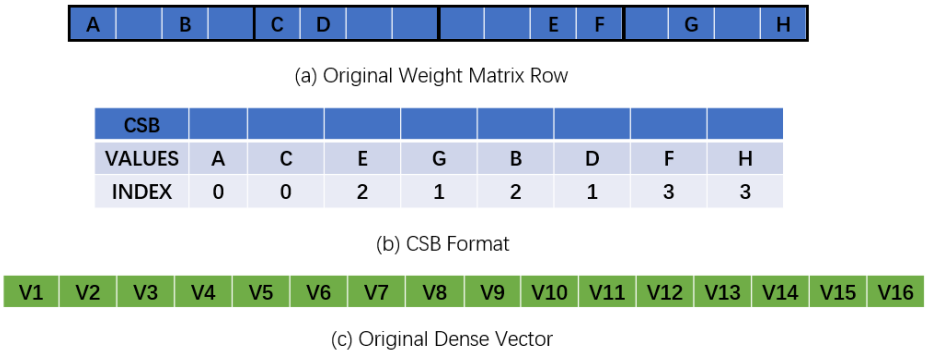


Figure 4.4: The Original Parameters

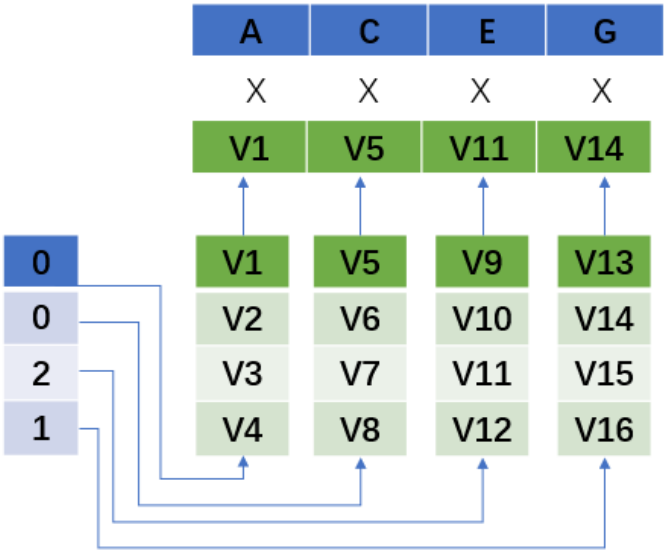


Figure 4.5: Inter-Bank Multiplication

In Figure4.4, there is one sparse matrix row and its CSB format. This row includes 4 banks with the size of 4 elements. We can see that the non-zero elements and their internal bank indices is already arranged by the order of banks. And in

(c), there is a dense vector. In Figure 4.5, the dense vector is split into 4 banks as the matrix row, and the indices from CSB are used to find the vector elements mapping weight elements from each bank(i.e. weight representation in the CSB format). Therefore inter-bank parallelism in dot production operations can be explored. This operation is realized in Private Vector Buffer(PVB), as in Figure 4.3.

EWOP Unit

EWOP unit is the unit to realize element-wise operations. The result vector from SpMxV unit is fed into this unit. The detailed computations of EWOP unit is as Figure 4.6.

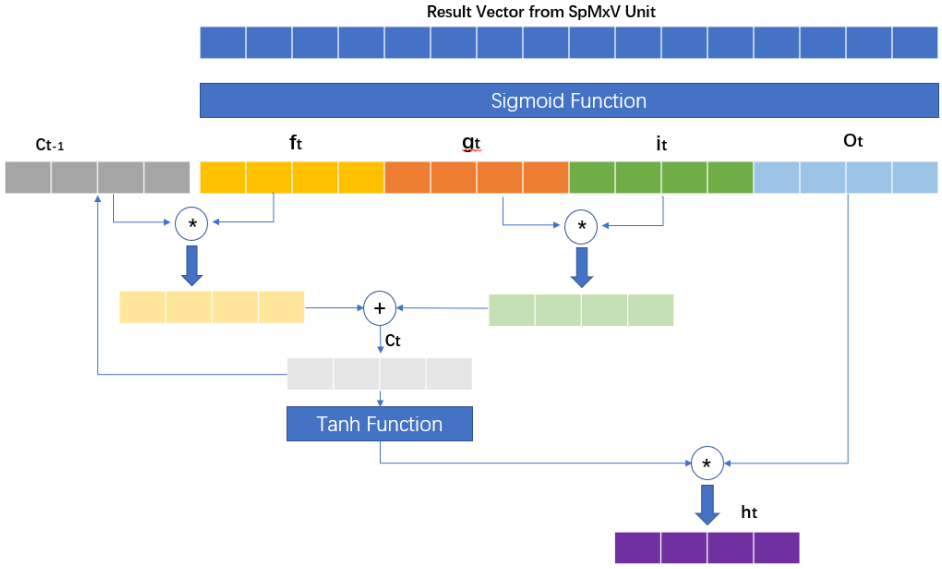


Figure 4.6: Element-wise Computations

Inside this unit, there are two activation functions: Sigmoid Function and Tanh Function. The result vector from SpMxV unit first is fed into Sigmoid Function, then split into different gate values. The splitting can directly perform because the location of weight matrices corresponding to each gates are fixed. After that, element-wise addition and multiplications are performed and then cell state will update after processed by Tanh Function. The final result h_t will go to Vector Memory, where merging new vectors for next computation.

In particular, to make sure the design is hardware-friendly, the activation function is implemented using piece-wise linear approximation[44]. The curve of activation functions are approximated with lines by equal-size interval. Each

piecewise linear interpolation can be represented as $y = a * x + b$. The length of interval is pre-defined, and these parameters a and b are computed by MATLAB and will be stored into activation funciton modules in LSTM. The approximation representation of sigmoid function and tanh function are as following:

- **Sigmoid Function**

$$f(x) = \begin{cases} 0 & x \leq -8 \\ a_{\left\lfloor \frac{x+8}{k} \right\rfloor} x + b_{\left\lfloor \frac{x+8}{k} \right\rfloor} & 0 < x \leq 8 \\ 1 & x > 8 \end{cases} \tag{4.1}$$

- **Tanh Function**

$$f(x) = \begin{cases} -1 & x \leq -6 \\ a_{\left\lfloor \frac{x+6}{k} \right\rfloor} x - b_{\left\lfloor \frac{x+6}{k} \right\rfloor} & -6 < x \leq 0 \\ a_{\left\lfloor \frac{x+6}{k} \right\rfloor} x + b_{\left\lfloor \frac{x+6}{k} \right\rfloor} & 0 < x \leq 6 \\ 1 & x > 6 \end{cases} \tag{4.2}$$

Controller

There is also a small controller, to sequence the computation flow of LSTM. There are three basic instructions received by controller from host server:

- **LOAD MEMORY** This instruction is performed before running the LSTM acceleration. It includes instruction **LOAD WEIGHTS**, **LOAD BIAS** and **LOAD VECTOR**. When controller receives this kind of instructions, it will send control signals to Matrix Memory and Vector Memory to ask them to load CSB values, CSB indices, bias and vectors from outsides, to prepare the execution of acceleration.
- **READ PARAMETERS** This instruction is used to ask LSTM accelerator to read weights, indices, vectors and biases from memories and execute the computations, as similar with a "start" switch.
- **NO READ** Because LSTM is a kind of recurrent neural network, the output will be used as next input. Therefore, when LSTM executing the computation, no data reading is needed. This instruction is used to closed the reading from memories.

Moreover, control unit also receives a status signal from EWOP unit, which means that EWOP output is available. Then control unit will send one signal to Vector Memory to enable the reading from EWOP.

4.5 C++ design

Based on Vivado HLS, the architecture described in previous section is written by C++ language, focusing on the behaviors of each module. The main module includes Matrix Memory, Vector Memory, SpMxV Unit, PE unit, EWOP unit, Control Unit, Sigmoid and Tanh Function and the top-level abstraction BBS Accelerator architecture, mapping main part of the architecture. As mentioned before, when performing high-level synthesis, a hierarchy of sub-functions will transform to a hierarchy of modules or entities in final RTL design. And these module written by C functions will synthesize as blocks in RTL.

Inside each module, multiple loops are used. That is because of the benefits of HLS which can easily explore parallelism by adding different optimization directives. By default the loops are realized by the logic for one iteration. And with directives such as unroll or pipeline, it can copy more logic implementation of one iteration or insert registers to achieve pipeline. The factor of unroll and initiation interval of pipeline can be set to different values.

The augments of top-level function will synthesize into RTL I/O port, and if any array exists in top-level function, it will synthesize into ports to access external BRAM.

Moreover, Vivado HLS provides a number of arbitrary precision data-types, so that users can decide arbitrary width in their design. This is one of advantages of HLS because in hardware more accurate bitwidths are usually used instead of the data types such as 32-bit int provided by C/C++. The arbitrary precision data types are defined in header files as Figure 4.7.

```
typedef float din_t;
typedef int   dindx_t;
typedef float dout_t;

typedef int cin_t;
typedef int cout_t;
```

Figure 4.7: The Definition of Arbitrary Precision Data Types

In this figure, 32-bit float types are used. Thanks to HLS, it is convenient to modify them with the different types and find the best accuracy bitwidth for specific hardware design.

The C codes of the design of this thesis are shown as Appendix A.

4.6 Vivado HLS

This thesis implements the design based on Vivado HLS. Then main steps in HLS design flow are:

- Validate C design by simulation
- Create a synthesis solution
- Verify the RTL by co-simulation and export IP packages

Before synthesis, C validation is performed by a C testbench in Vivado HLS C debug environment, to check the correctness of the design. A new project is created by script run_hls.tcl, shown as B. The windows of Vivado HLS is as Figure 4.8.

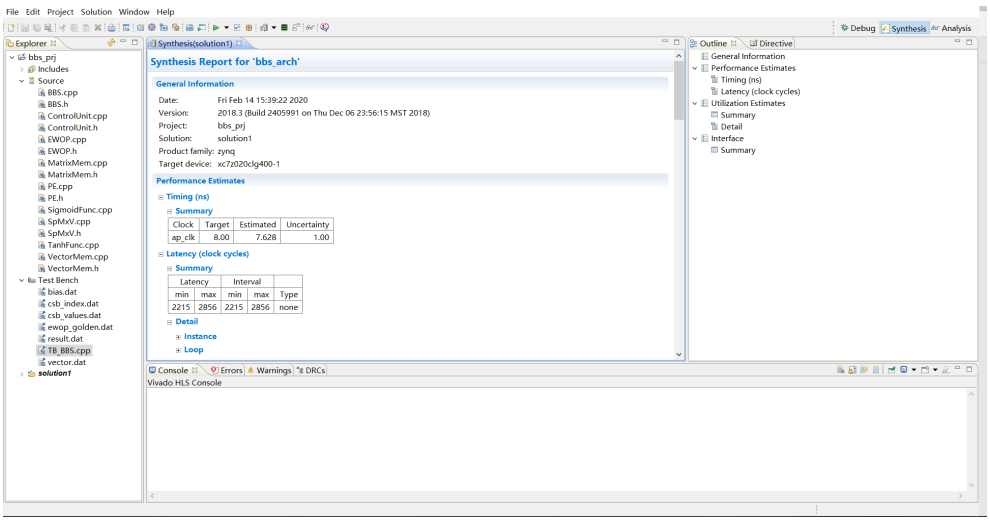


Figure 4.8: The Window of Vivado HLS Project

All of C design codes are in source folder and testbench is in Test Bench folder. To perform the simulation, the data files also need to added into Test Bench folder. After running C simulation, a csim.log will be shown automatically, to show the simulation result. Next, the synthesis is performed. when it finished, a report opens and performance estimates are shown. At this step, RTL files in both verilog and VHDL are generated.

In particular, before running C synthesis, the directives tab on the right shows the objects in the design that can be optimized, such as for-loops. Here different directives can be added by two ways: one is directly added on the source codes; another is added these directives into a script called directives.tcl. With former way, the directives can be kept with source files without needing scripts when re-use

these design in other projects; and later way has the benefits to avoid changing source files.

To perform RTL verification, the C test bench is used again and generates inputs for RTL for simulation. The output from RTL are applied back into C test bench to check the correctness of result. After verifying RTL and result is pass, an IP block can be exported for using later in Vivado Design Suite.

4.7 IP Connection

After exporting the design IP, we need incorporate the design into the FPGA. This step is operated in the Vivado Design Suite. When creating new project, we should select target devices. In this project, we used Xilinx PYNQ-Z2 board. The initial Project is shown as Figure 4.9.

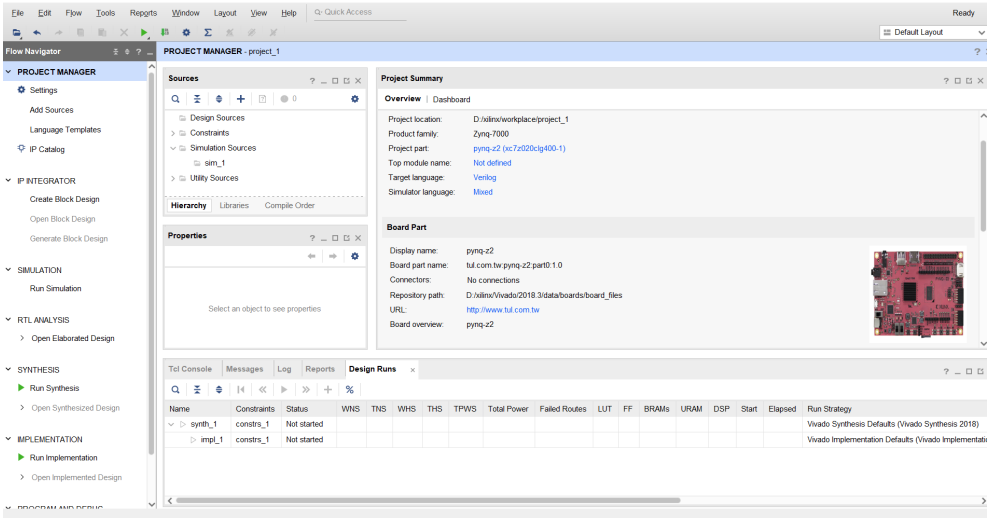


Figure 4.9: The Window of Vivado Design Suite

The main task in this project is to generate bitstream files for programming FPGA. The steps are following:

- Add HLS IP to the IP Catalog.
- Create an IP integrator block design of the system by connecting IP with Zynq processor.
- Implement the system by generating implementation sources and create an HDL wrapper as top-level module.

In particular, when creating the IP integrator, ZYNQ7 Processing System which is already in IP catalog is used, and users need to preset this block in order to connect their design with processor. In this project, to match the interface of designed IP, the overall presetting is shown as Figure4.10.

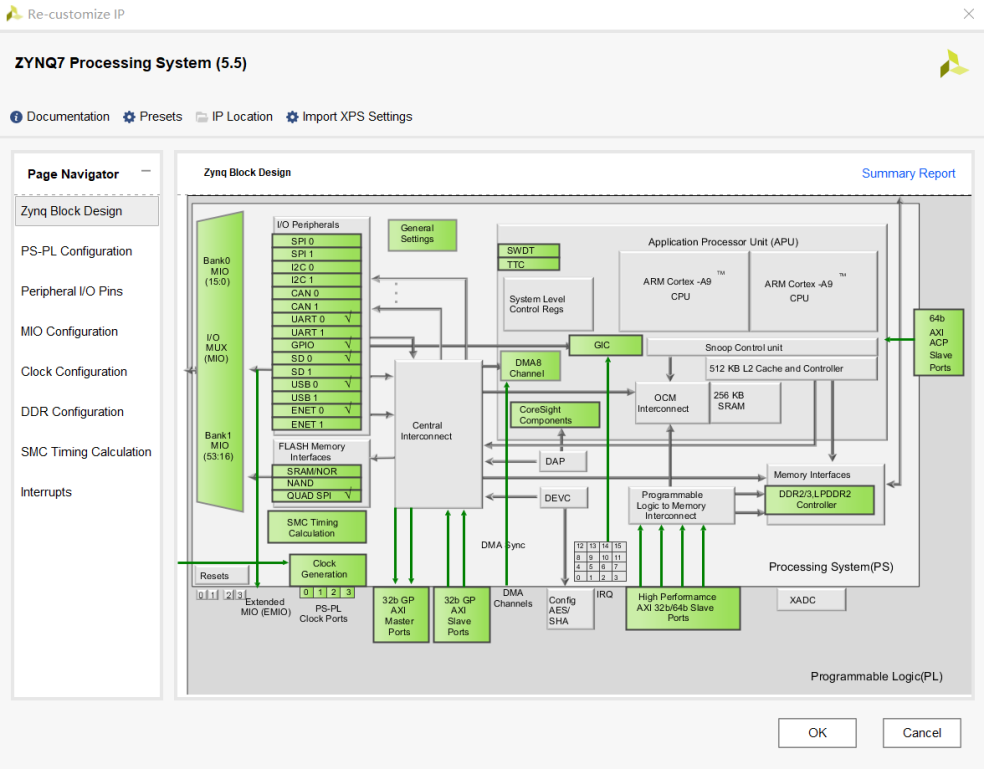


Figure 4.10: The Presetting of ZYNQ7 Processing System

The IP connection with designed IP core is as Figure 4.11.

Here the block bbs_arch0 is the designed HLS IP. It is connected with Processing System by AXI Interconnect. A Proc Sys Reset block is also generated. After connection, validating design is performed. When the validation is successful, generating output products is performed to obtain source files. And then we should create HDL wrapper to make top-level of design becomes a verilog file. It means the design is ready to be synthesized and implemented to generate bitstream files for FPGA programming. After generating bitstream files successfully, SDK tools are used to program FPGA.

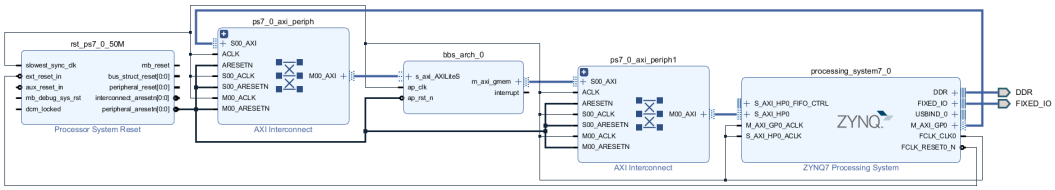


Figure 4.11: The IP Connection

4.8 Running Design on FPGA

This section is working on Xilinx SDK, where allows users to create software driver to run their design on target devices. The SDK environment is as Figure 4.12.

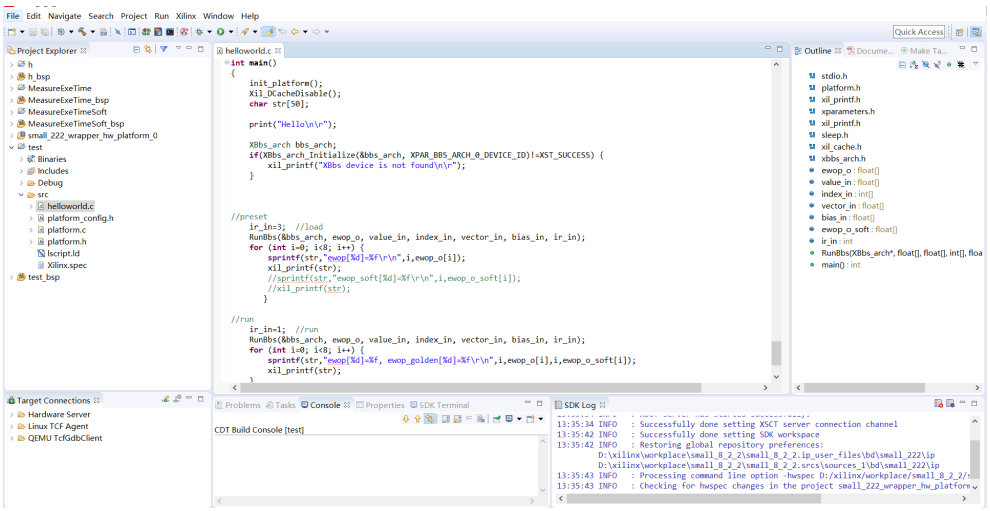


Figure 4.12: The SDK Tools

After exporting hardware including bitstream file in Xilinx Design Suite, we need to launch SDK and create a new project. This project is automatically based on the target device environment. When creating a new application, first we need to check the programming and communication with FPGA. Therefore, a Hello World

template is used. After connecting the board with PC and set it up, programming FPGA is performed. At the same time, Putty is used as a monitor to check the communication.

Next, a software driver is written to communicate with HLS block, shown in AppendixC. This driver is written by C language. Serveral BSP and C header files are included. In BSP, the corresponding functions such as reading/writing data are generated by tools, to support the software driving. Then after running the software, the result returned by FPGA is shown on Putty, as Figure 4.13.

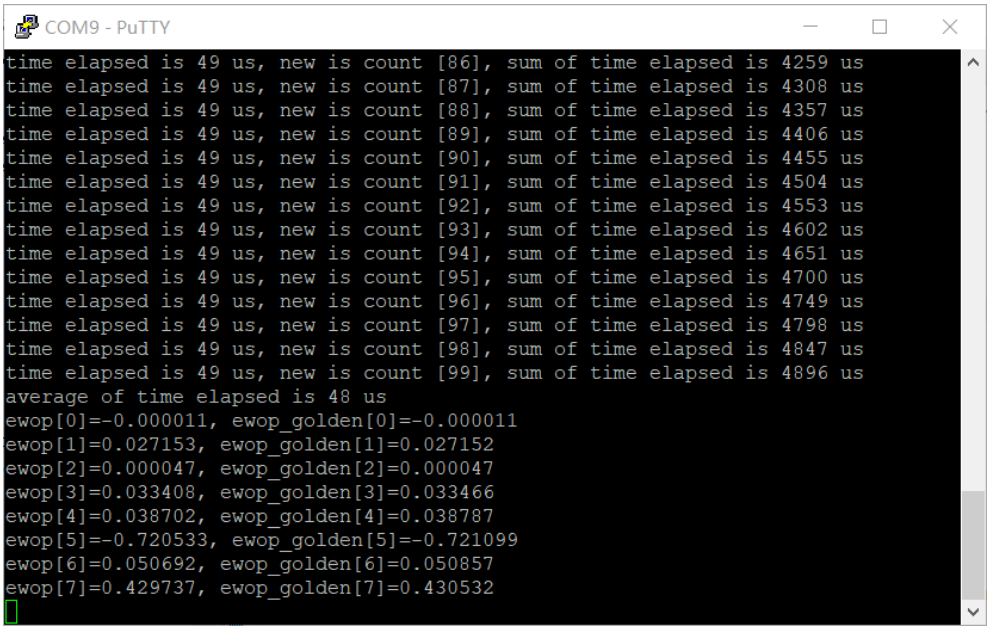


Figure 4.13: Result from FPGA

To verify the result, a golden value data file is generated by MATLAB. This result computing by performing the matrix-vector multiplication directive and use activation functions without any approximation. The result of our design is almost same comparing to golden values, the errors caused by linear approximation of activation functions only at 0.1%.

4.9 Implementation with AXI DMA

Moreover, in this thesis, one implementation with AXI Direct Memory Access (DMA) peripheral also is proposed. In this solution, the hardware accelerator IP core is connected to the Accelerator Coherency Port (ACP) of the ARM CPU in the SoC device with AXI4-Stream interface. A DMA core in the SoC Programmable Logic (PL) subsystem is used to make the connection. It can provide high-bandwidth

direct memory access between memory and peripherals. Then the AXI DMA is connected to the L2 cache of the ARM processor, which achieves a higher throughput in data communication. The block diagram of the system is shown as Figure 4.14 [45].

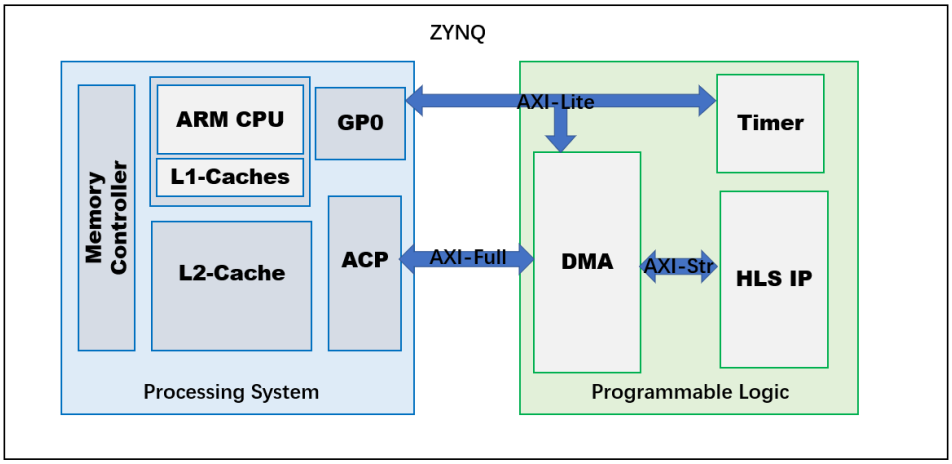


Figure 4.14: Block Diagram of Communicating with DMA

4.9.1 AXI4-Stream Interface and ARM ACP

To use AXI DMA for communication, the interface of the design need to be implemented with AXI-Stream communication standard. It is a kind of free-addressing point-to-point data transfer protocol. To connect our hardware accelerator design to AXI DMA, pop stream and push stream functions are written in C++ codes, to support the data extraction and insertion from/into an AXI4-Stream interface, and also change the data type into 32-bit unsigned data of AXI4 protocol. The pop stream function and push stream function are shown as Figure 4.15:

To use AXI4-Stream interface, some side channel information associated should be provided by user. Therefore, a data type defined as AXI_VAL written in C++ code. In the C testbench, the pop stream function is used to insert data into AXI Stream and push stream function is used to extract result returned by the accelerator from AXI Stream.

The Accelerator Coherency Port (ACP) of the ARM CPU is a 64-bit AXI slave interface. This interface makes asynchronous cache-coherent access directly from SoC PL to Cortex-A9 CPU processor subsystem achievable, provides a low latency path [45]. Through the ACP port, the processor L1 caches are searched for the required information, if it misses, the L2 cache is accessed, and finally the main

```
template <typename T, int U, int TI, int TD>
T pop_stream(ap_axiu <sizeof(T)*8,U,TI,TD> const &e)
{
#pragma HLS INLINE

//  assert(sizeof(T) == sizeof(int));
    union
    {
        int ival;
        T oval;
    } converter;
    converter.ival = e.data;
    T ret = converter.oval;

    volatile ap_uint<sizeof(T)> strb = e.strb;
    volatile ap_uint<sizeof(T)> keep = e.keep;
    volatile ap_uint<U> user = e.user;
    volatile ap_uint<1> last = e.last;
    volatile ap_uint<TI> id = e.id;
    volatile ap_uint<TD> dest = e.dest;

    return ret;
}
```

```
template <typename T, int U, int TI, int TD> ap_axiu
<sizeof(T)*8,U,TI,TD>
push_stream(T const &v, bool last = false)
{
#pragma HLS INLINE
    ap_axiu<sizeof(T)*8,U,TI,TD> e;

//  assert(sizeof(T) == sizeof(int));
    union
    {
        int oval;
        T ival;
    } converter;
    converter.ival = v;
    e.data = converter.oval;

    // set it to sizeof(T) ones
    e.strb = -1;
    e.keep = 15; //e.strb;
    e.user = 0;
    e.last = last ? 1 : 0;
    e.id = 0;
    e.dest = 0;
    return e;
}
```

Figure 4.15: Pop Stream and Push Stream

memory if miss happens again.

4.9.2 AXI DMA

AXI DMA is a kind of IP core in Vivado Repository, its block diagram is shown as Figure 4.16 [45].

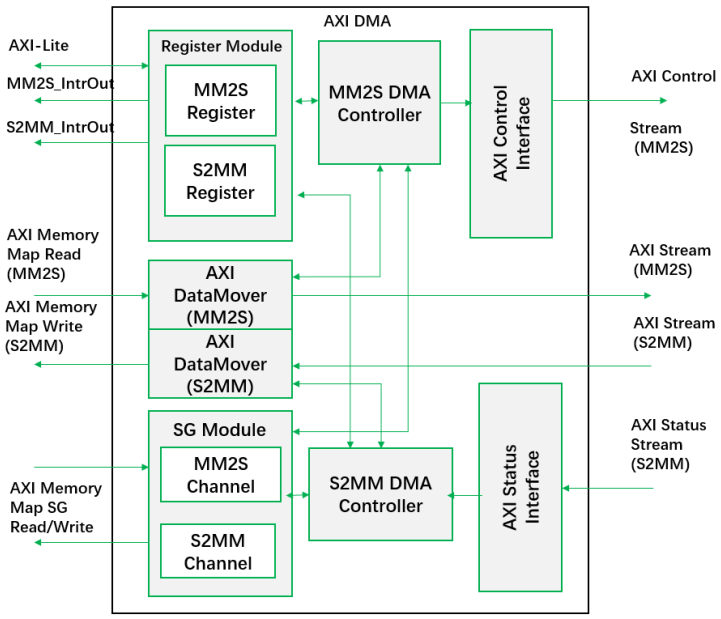


Figure 4.16: AXI DMA

This DMA core can handle address generation, burst formatting and scheduling of memory transaction [45]. The communication is achieved by a series of AXI4 interfaces, as shown in the block diagram. In particular, MM2S means AXI4 to AXI4-Stream and S2MM means AXI4-Stream to AXI4. In this thesis, the DMA works under Simple DMA mode, which requires less resource utilization.

In this DMA mode, AXI Memory Map Read (MM2S) interface reads data from external memory and then the data are transmitted by DataMover to slave through the AXI4-Stream (MM2S) port. On the another hand, AXI Memory Map Write (S2MM) interface can receive data and send them to a slave external memory through the AXI4-Stream (S2MM) port.

4.9.3 IP Integrator Design

As described in the previous section, after exporting IP core, Vivado Design Suite is used to create the hardware design targeting the PYNQ-Z2 board. The block design is shown as Figure 4.17.

In this block diagram, bbs_arch_0 is our hardware accelerator IP core. The input para_in is connected with AXI DMA to the interface called M_AXIS_MM2S,

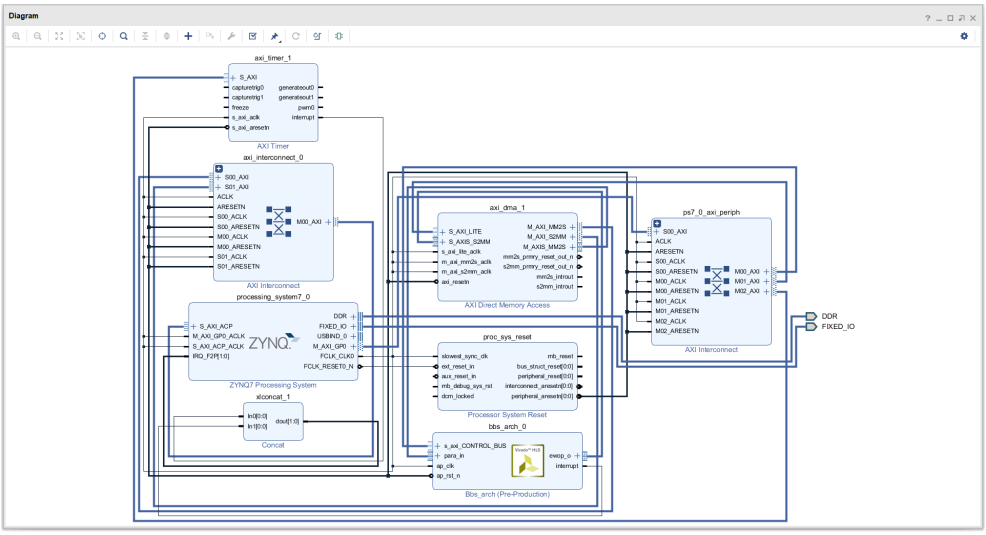


Figure 4.17: IP Connection with DMA

which means the AXI4-Stream Master to Slave. And the output ewop_o is connected with AXI DMA to the interface called S_AXIS_S2MM, which means the AXI4-Stream Slave to Master. In other words, the Programmable Logic (PL) read/write DMA data through AXI-S interface. The Processing System (PS) sends instructions into AXI DMA by AXI-Lite interface.

Moreover, the Address Editor shows the memory map and unmapped slaves should be assigned suitable address. The address assignment is shown as Figure 4.18.

Cell	Slave Interface	Base Name	Offset Address	Range	High Address
axi_dma_1					
Data_MM2S (32 address bits : 4G)					
processing_system7_0	S_AXI_ACP	ACP_DDR_LOWOCM	0x0000_0000	512M	0x1FFF_FFFF
processing_system7_0	S_AXI_ACP	ACP_IOP	0xE000_0000	4M	0xE03F_FFFF
processing_system7_0	S_AXI_ACP	ACP_M_AXI_GP0	0x4000_0000	1G	0x7FFF_FFFF
processing_system7_0	S_AXI_ACP	ACP_QSPI_LINEAR	0xFC00_0000	16M	0xFCFF_FFFF
Data_S2MM (32 address bits : 4G)					
processing_system7_0	S_AXI_ACP	ACP_DDR_LOWOCM	0x0000_0000	512M	0x1FFF_FFFF
processing_system7_0	S_AXI_ACP	ACP_IOP	0xE000_0000	4M	0xE03F_FFFF
processing_system7_0	S_AXI_ACP	ACP_M_AXI_GP0	0x4000_0000	1G	0x7FFF_FFFF
processing_system7_0	S_AXI_ACP	ACP_QSPI_LINEAR	0xFC00_0000	16M	0xFCFF_FFFF
processing_system7_0					
Data (32 address bits : 0x40000000 [1G])					
axi_dma_1	S_AXI_LITE	Reg	0x4040_0000	64K	0x4040_FFFF
axi_timer_1	S_AXI	Reg	0x4280_0000	64K	0x4280_FFFF
bbs_arch_0	s_axi_CONTROL_BUS	Reg	0x43C0_0000	64K	0x43C0_FFFF

Figure 4.18: Address Assignment

After generating the bitstream based on this IP integrator design, we launch

SDK and create the software driver for supporting the hardware accelerator.

Chapter 5

Experimental Results

5.1 Experimental Setup

In this thesis, development tools from the Xilinx Vivado Design Suite have been used, including Vivado HLS, Vivado and Vivado SDK, which have been discussed in previous sections. The hardware platform is Xilinx PYNQ-Z2, which is a Zynq development board designed to be used with a open-source framework.

5.1.1 Hardware Platform

The implementation and evaluation of the design in this work has been performed on the development board PYNQ-Z2. With PYNQ open-source framework, Xilinx Zynq All Programmable SoC(APSoC) function supports user to create their design just focusing on the programming. The core of PYNQ-Z2 is ZYNQ XC7Z020 FPGA, which including 650MHz ARM Cortex-A9 dual-core processor and multiple programmable logic. PYNQ-Z2 can also support Python, can achieve on-line programming and debugging.

To setup the board, a Micro SD card loaded with the PYNQ-Z2 image needs to insert. After connecting the USB cable and Ethernet cable, the board can be turned on. Moreover, the Ethernet cable can be connected to a router or directly a PC. When connecting to a computer, static IP address should be assigned for the computer, in order to communicate with board. The IP address of the board is 192.168.2.99, so that the static IP assigned for computer should be at the same range as the board.

After turning on the board, the board will load the image from Micro SD card. Once two Blue LEDs and four Yellow LEDs flash, the system is booted and ready to use.

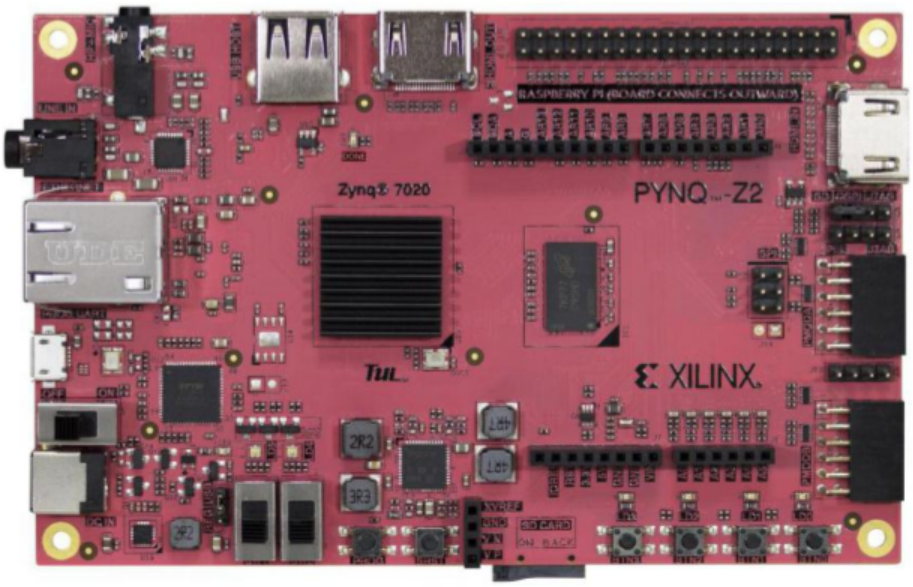


Figure 5.1: PYNQ-Z2

5.2 Evaluation

In this thesis, there are three hardware accelerators with different hidden layer size are implemented, shown as Table 5.1.

Hidden Size	Matrix Size	Bank Size
8	32×16	4
32	128×64	8
64	256×128	16

Table 5.1: Implementation of Accelerator with Different Size

These implementation are also optimized by setting directives. Moreover, the implementation with DMA and without DMA are also evaluated. The main parts of evaluation are shown as following:

- Speedup of unoptimized and optimized implementations
- Resource utilization of unoptimized and optimized implementations
- Power consumption of unoptimized and optimized implementations

- Comparison between implementation with DMA and the one without DMA, under the same hidden layer size and directives
- Pareto curve of different solutions

5.2.1 Speedup

As the goal of accelerating LSTM models, speedup is one of main evaluations. In this part, we set average of execution time measured 1000 times as the speed to perform comparisons. Three accelerators with 8-size, 32-size and 64-size hidden layers are implemented respectively. In particular, unoptimized implementation and optimized implementation are evaluated in this part. All of these sparse LSTM accelerators are set with sparsity equal to 50%.

These accelerators can also be used for larger LSTMs, by splitting the input vector and weight matrices appropriately. This however will have a cost in terms of speed-up, since the new portion of the weight matrix has to be reloaded after each invocation of the acceleration.

The reference is the average execution time of software implementation on the ARM Cortex-A9 processor present on the Zynq SoC. The speedup ratio is shown as Figure 5.2.

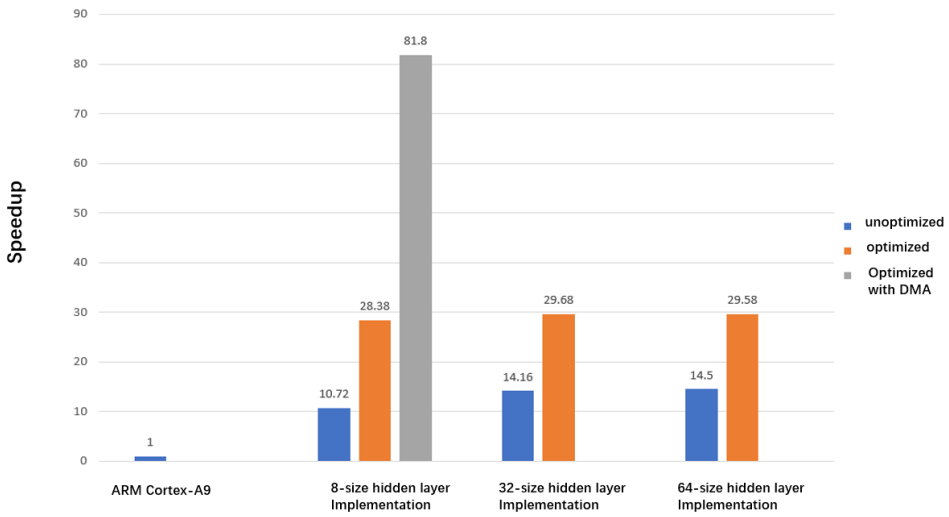


Figure 5.2: Speedup

As show in the figure, for small accelerator version (8-size hidden layer one), a 11x speedup has been obtained compared to a software implementation on the ARM Cortex-A9 processor present on the Zynq SoC; when scaling to bigger

designs (32-size hidden layer and 64-size ones), the speedup increases to 15x. This is expected since, as explained in Section 2, the number of dot products grows quadratically with the hidden size.

Moreover, by setting some directives, the speedup increases to 30x. It is expected because directives are used to perform optimization, such as latency can be reduced by adding some pipeline register and so on.

In this part, the 8-size hidden layer optimization implementation with DMA and the one without DMA are compared. We can see that when we perform optimization by setting directives, the speedup increases to 28x from 11x; with using AXI DMA, the speedup sharply increases to 82x, which totally shows the performance improve by reducing data transfer latency.

5.2.2 Resource Utilization

The resource utilization is evaluated by the utilization of BRAM, DSP48E, FF and LUT. The evaluation is shown as Table 5.2.

Hidden Size	BRAM	DSP48E	FF	LUT
8 (without optimization)	15	34	20997	23868
8 (with optimization)	21	52	59374	51925
8 (with optimization and DMA)	25	45	43859	41197
32 (without optimization)	47	34	20500	23994
32 (with optimization)	52	52	58666	52812
64 (without optimization)	143	34	20546	24088
64 (with optimization)	148	36	38693	49007

Table 5.2: Resource Utilization

In this table, we can see that performing optimization under the same size of implementation, the BRAM and DSP48E resources are increase will little, and FF and LUT resources are doubled. By using the optimization methods such as pipeline, the computing resources of the FPGA are used efficiently, in order to increase speed as shown in previous section.

One of the interesting point is that the optimized implementation with DMA, which achieve 82x speedup as shown in previous section, even use less resources than the implementation without DMA under the same optimization.

The resource utilization is also evaluated by comparing our design with one dense LSTM implementation[9] from Zhang et al. Both LSTM accelerator is implemented to support 32-size hidden layer. The comparison is shown as Figure 5.3.

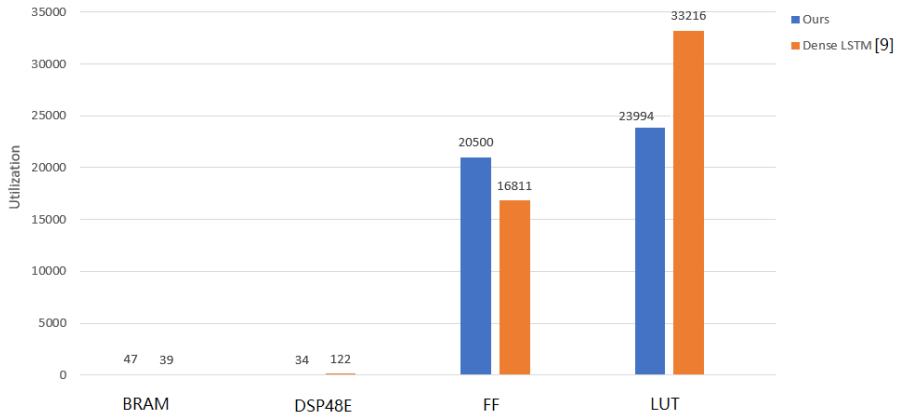


Figure 5.3: Resource Utilization

The resource utilization is evaluated by the utilization of BRAM, DSP48E, FF and LUT. Considering the different hardware platform, the utilization is not represented by percentage of utilization but directly shown the number. As we can see in the figure, the utilization of BRAM resource is similar in both implementation, but for the utilization of DSP48E, FF and LUT, the sparse LSTM design shows better area-saving feature.

5.2.3 Power consumption

Power consumption of LSTM accelerator is also evaluated by power report in Vivado Design Suite. To analysing the power obtained, accelerator unoptimized and optimized implementations with 8-size, 32-size and 64-size hidden layers are compared.

In this figure, we can see that when the layer of LSTM growing from 8-size to 32-size and 64-size, the increasing of power consumption is very small, almost negligible. When comparing with the dense LSTM with same size(32-size) from Zhang et al., power consumption of our design is lightly smaller.

Moreover, with optimization and without optimization under the same size, the power consumption is only lightly increase. When using DMA under the same size and optimization, the power consumption even smaller. The power consumption of all implementation prove the lower power consuming feature of FPGA.

5.2.4 Pareto Curve

Based on HLS, the accelerator design has a flexibility to perform different optimizations by adding directives, to force the implementation with different versions.

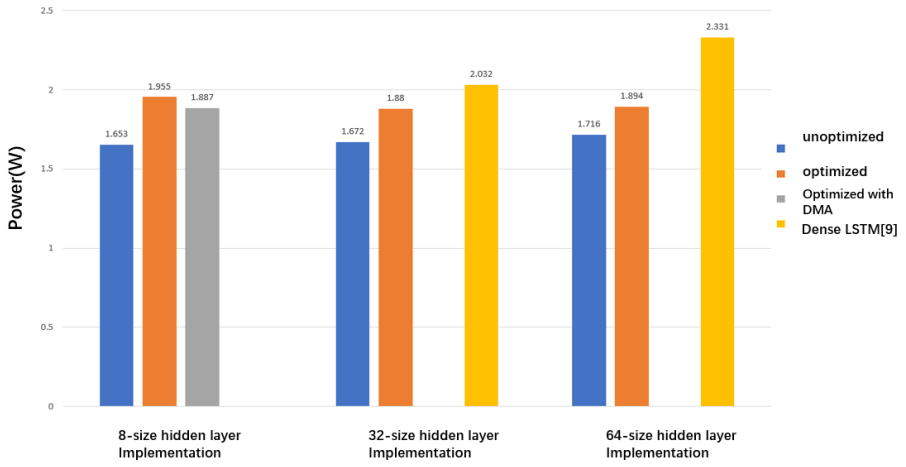


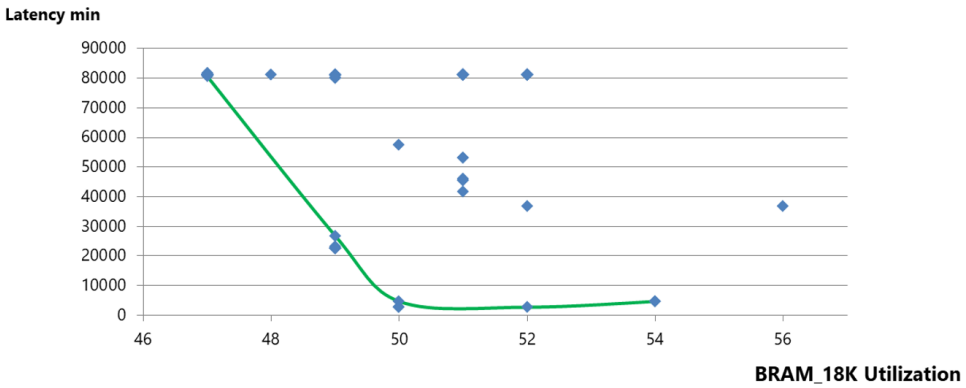
Figure 5.4: Power Consumption

Therefore, the design space exploration can be achieved to find the suitable design implementation for different specific application.

In this part, the hardware accelerator with 32-hidden layer size are explored different implementations by setting different optimization directives. The both cases with DMA and without DMA are considered.

To obtain the relationship between speed and area, both minimum latency and maximum latency are considered, as the reference to represent speed. With higher latency, the speed is slower. And the area is represented by four resource utilization: BRAM, DSP48E, FF and LUT. In each case, the pareto curve is shown.

Without DMA, the different implementation of hardware accelerator with 32-hidden layer size are realized. With reference as minimum latency and maximum latency, the pareto curves are shown as Figure 5.5 and 5.6 respectively.



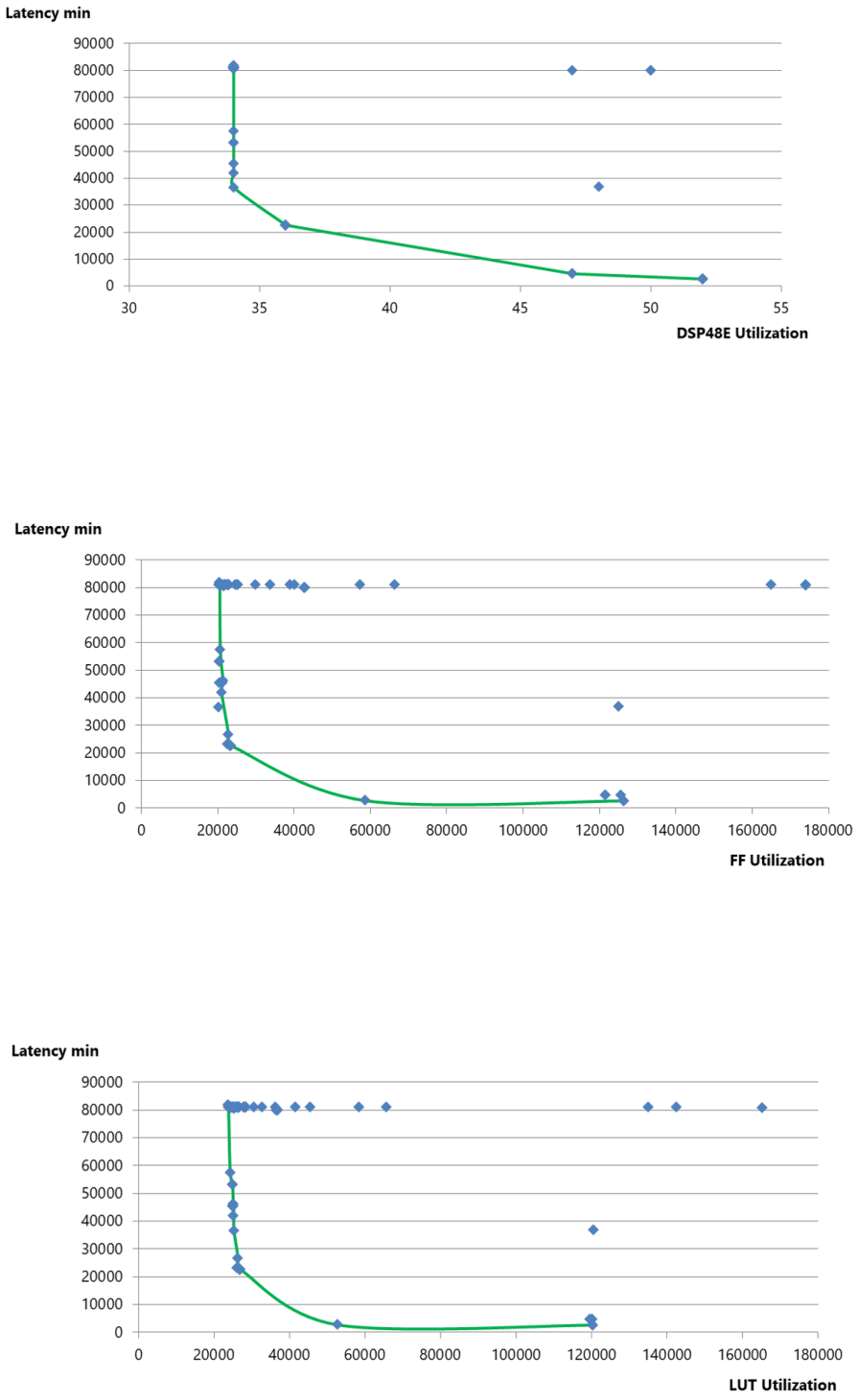
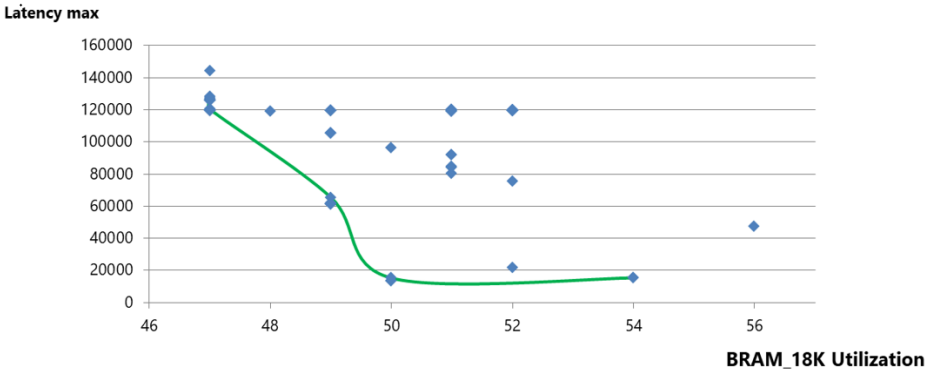
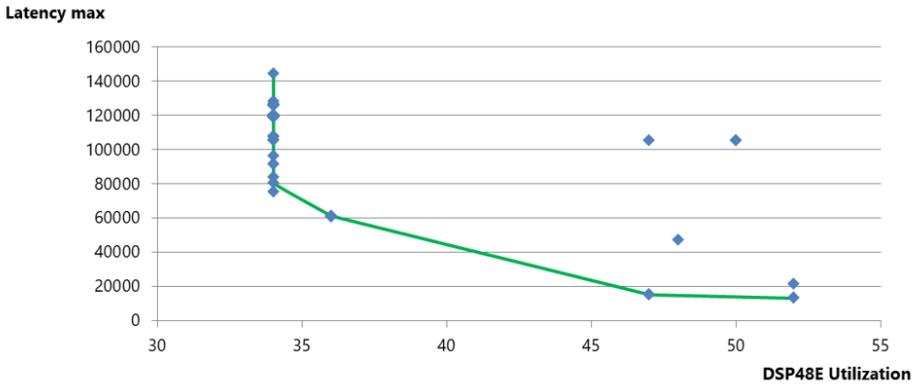
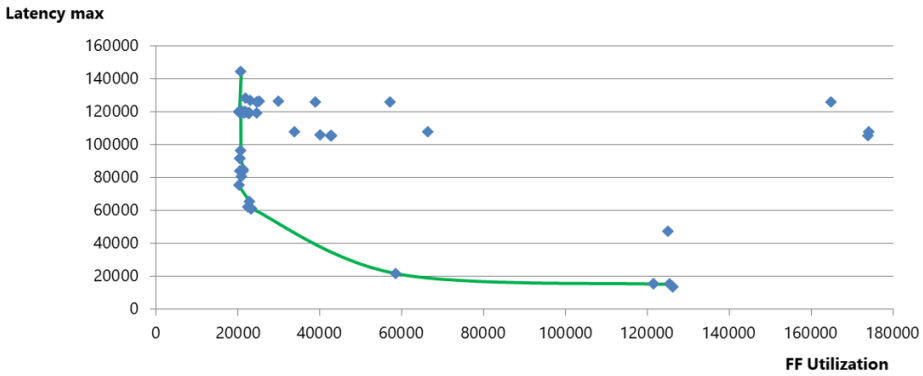


Figure 5.5: Pareto Curves under 32-hidden Layer Size with Minimum Latency


Latency max
160000
140000
120000
100000
80000
60000
40000
20000
0
46
48
50
52
54
56
BRAM_18K Utilization


Latency max
160000
140000
120000
100000
80000
60000
40000
20000
0
30
35
40
45
50
55
DSP48E Utilization


Latency max
160000
140000
120000
100000
80000
60000
40000
20000
0
0
20000
40000
60000
80000
100000
120000
140000
160000
180000
FF Utilization

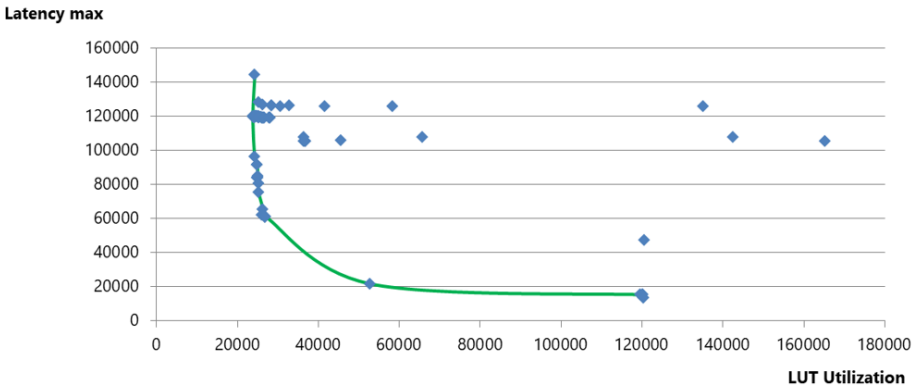


Figure 5.6: Pareto Curves under 32-hidden Layer Size with Maximum Latency

In these figures, with multiple design implementation based on optimization directives, we can find some solutions with higher latency (slower) but smaller area (smaller resource utilization), and also some ones with lower latency but bigger area. This exactly proves the benefits of HLS that designers can provide numerous solutions to users by short development period, and the best solution under requirements can be explored without changing source design.

Thanks to HLS, in this case, we are able to obtain implementations with a worst-case minimum latency ranging from about 81000 clock cycles to about 2500 clock cycles, and maximum latency ranging from about 144325 to 13000. The number of BRAM varies from 47 to 52, and of DSP48E varies from 31 to 52, not changing too much. The number of FF varies from 20317 to 174172, and of LUT varies from 23797 to 165331.

These Pareto curves are for the accelerator without DMA. However, notice that the Pareto curve will be very similar also for the DMA case. In fact, most of the internal accelerator architecture will be the same, and the presence of DMA will only influence the interface part. In other words, the larger speed-up obtained in the DMA case is not due to difference in the accelerator latency, but just in the input/output data transfer throughput.

Chapter 6

Conclusion and Future Works

6.1 Conclusion

Hardware accelerator, as a popular solution to speed up the computation process of LSTM models, attracts many researchers to explore noval algorithms and architectures. Focusing on the numerous parameters inside the model, sparse LSTM networks are proposed with various pruning method and compressed sparse formats. Moreover, further to solve the unstructured matrix problem, constrained pruning is considered, and corresponding hardware architectures are created.

On the anther hand, with high performance and low power consumption, FPGA becomes a good choice as hardware platform to support LSTM models. Recently, since High level syntheis(HLS) techniques becomes more and more mature. With the flexibility of programming and optimization, a various LSTM accelerator implementation can be achieved, and the exploration of design space becomes much faster.

In this thesis, a hardware accelerator supporting LSTM models based on FPGA is designed and implemented by HLS techniques. Focusing on the resource saving, Bank-Balanced Sparse(BBS) pruning is used, which solves unstructured sparse matrix and irregular memory access problems. At the same time, Compressed Sparse Bank(CSB) format is used, without introducing encoder overhead.

Based on HLS, both C++ designs using DMA and without DMA are synthesized into RTL and finally implemented on the Xilinx PYNQ-Z2 FPGA board. The evaluations on speedup, resource utilization and power consumption are performed. Finally, a complete design space exploration has been performed and the pareto curves have been found. Many different accelerator implementations have been realized, among which a system designer could select depending on his/her

requirements, by analysis of the speed-area relations of different designs.

6.2 Future Work

This LSTM accelerator can be further optimized. In this work, the accelerator is implemented with 32-bit float point data type. By using shorter data types such as fixed-point, the size of LSTM accelerator could be further reduced at the cost of some additional accuracy loss. Moreover, based on the sparsity of output of EWOP unit(in particular, h_t vector), the pruning applied to h_t could be explored, which will lead the computational costs further reducing.

In the future work, focusing on the neural networks with huge scale, the accelerator could be interfaced with the processor using multiple DMA to achieve faster data transfers during runtime. This would allow to fully exploit the parallelism inside the FPGA.

Appendix A

C++ Design

A.1 Top-Level Function

A.1.1 Header File

```
#ifndef _BBS_H_
#define _BBS_H_

#include "stdio.h"
#include "ControlUnit.h"
#include "MatrixMem.h"
#include "VectorMem.h"
#include "SpMxV.h"
#include "EWOP.h"

//matrix mem
#define ARRAY_SIZE 128 //value and index size(csb)

//vector mem
#define VECTOR_SIZE 16
#define BIAS_SIZE   16  //from vector memory, equal to the #of rows in matrix
#define GATE_SIZE   4

//SpMxV
#define OUT_SIZE    16

typedef float din_t;      //define the data type
typedef int   dindx_t;
typedef float dout_t;

```

```
26 typedef int cin_t;
27 typedef int cout_t;
28
29 void bbs_arch (dout_t ewop_o[GATE_SIZE*2], din_t value_i[ARRAY_SIZE],
       dindx_t index_i[ARRAY_SIZE], din_t vector_i[VECTOR_SIZE], din_t
      bias_i[BIAS_SIZE], cin_t ir_in);
30
31 #endif
```

A.1.2 BBS.cpp

```
1 #include "BBS.h"
2
3 void bbs_arch (dout_t ewop_o[GATE_SIZE*2], din_t value_i[ARRAY_SIZE],
       dindx_t index_i[ARRAY_SIZE], din_t vector_i[VECTOR_SIZE], din_t
      bias_i[BIAS_SIZE], cin_t ir_in) {
4
5 //control singals
6     cin_t en_wr_v;     //status signal from ewop
7
8     cout_t rd_matrix;  //control signals from cu
9     cout_t rd_vector;
10    cout_t wr_vector;
11
12    cout_t ld_matrix;
13    cout_t ld_vector;
14
15 //matrix memory output, into spmxv unit
16    static dout_t matrix_value_o[ARRAY_SIZE];
17    static dindx_t matrix_idx_o[ARRAY_SIZE];
18
19 //vector memory
20    static dout_t vector_o[VECTOR_SIZE];
21    static dout_t bias_o[BIAS_SIZE];
22
23 //spmxv output
24    static dout_t spmxv_o[OUT_SIZE];  //after adding bias, into EWOP
25
26
27 //architecture
28    cu(&rd_matrix, &rd_vector, &wr_vector, &ld_matrix, &ld_vector,
      ir_in, &en_wr_v);
29
30    matrix_mem(matrix_value_o, matrix_idx_o, value_i, index_i, &
      ld_matrix, &rd_matrix);
31
```

```
32      vector_mem(vector_o, bias_o, vector_i, bias_i, ewop_o, &ld_vector
        , &rd_vector, &wr_vector);
33
34      spmxv(spmxv_o, matrix_value_o, vector_o, matrix_idx_o, bias_o);
35
36      ewop(ewop_o, &en_wr_v, spmxv_o, ir_in);
37
38  }
```

A.2 Matrix Memory

A.2.1 Header file

```
1  #ifndef _MatrixMem_H_
2  #define _MatrixMem_H_
3
4  #include <stdio.h>
5
6  #define ARRAY_SIZE 128
7
8  typedef float din_t;
9  typedef float dout_t;
10 typedef int dindx_t;
11 typedef int cin_t;
12
13 void matrix_mem(dout_t value_o[ARRAY_SIZE], dindx_t idx_o[ARRAY_SIZE
       ], din_t value_i[ARRAY_SIZE], dindx_t idx_i[ARRAY_SIZE], cin_t *ld
       , cin_t *rd);
14
15 #endif
```

A.2.2 MatrixMem.cpp

```
1
2  #include "MatrixMem.h"
3
4  void matrix_mem(dout_t value_o[ARRAY_SIZE], dindx_t idx_o[ARRAY_SIZE
       ], din_t value_i[ARRAY_SIZE], dindx_t idx_i[ARRAY_SIZE], cin_t *ld
       , cin_t *rd) {
5
6
```

```
7  //CSB value and index
8      static dout_t value_current[ARRAY_SIZE];
9      static dindx_t idx_current[ARRAY_SIZE];
10
11 //if rst, load csb value and index data from files
12
13     if (*ld == 1) {
14 //reset value
15         Load_v_loop: for (int i=0; i<ARRAY_SIZE; i++) {
16             value_current[i]=value_i[i];
17         }
18 //reset index
19         Load_idx_loop: for (int j=0; j<ARRAY_SIZE; j++) {
20             idx_current[j]=idx_i[j];
21         }
22     //  int aaa=0;
23     }
24     else {
25         if (*rd==1) { //vector output
26         Out_loop: for (int i=0; i< ARRAY_SIZE; i++) {
27             value_o[i]=value_current[i];
28             idx_o[i]=idx_current[i];
29             }
30         }
31     }
32 }
```

A.3 Vector Memory

A.3.1 Header file

```
1
2  #ifndef _VectorMem_H_
3  #define _VectorMem_H_
4
5  #include "stdio.h"
6
7  #define ARRAY_SIZE_VECTOR 16
8  #define BIAS_SIZE_VECTOR  16
9  #define GATE_SIZE_VECTOR  4  //for each gate
10
11 //#define NUM_PE     8
12
13 typedef float din_t;
14 typedef float dout_t;
```

```
15 typedef int dindx_t;
16
17 //for control signals
18 typedef int cin_t;
19 typedef int cout_t;
20
21 void vector_mem (dout_t vector_o[ARRAY_SIZE_VECTOR], dout_t bias_o[
       BIAS_SIZE_VECTOR], din_t vector_i[ARRAY_SIZE_VECTOR], din_t bias_i
       [BIAS_SIZE_VECTOR], din_t ewop_o[GATE_SIZE_VECTOR*2], cin_t *ld,
       cin_t *rd, cin_t *wr);
22
23 #endif
```

A.3.2 VectorMem.cpp

```
1
2 #include "VectorMem.h"
3
4 //store the vector and bias
5 //control signals: rst:active low; rd: valid output; wr: valid input
6 //vector_o: output vector; bias_o: output bias
7 //ewop_o: result from ewop unit and generate new vector going to
      spmxv
8
9 void vector_mem (dout_t vector_o[ARRAY_SIZE_VECTOR], dout_t bias_o[
      BIAS_SIZE_VECTOR], din_t vector_i[ARRAY_SIZE_VECTOR], din_t bias_i
      [BIAS_SIZE_VECTOR], din_t ewop_o[GATE_SIZE_VECTOR*2], cin_t *ld,
      cin_t *rd, cin_t *wr) {
10
11     static dout_t vector_current[ARRAY_SIZE_VECTOR];
12     static dout_t bias_current[BIAS_SIZE_VECTOR];
13
14 //if rst, load vector and bias data
15 //for larger data set, need to read data while working ->DMA-
      uncompleted
16     if (*ld == 1) {
17         Load_vec_loop: for (int i=0; i< ARRAY_SIZE_VECTOR; i++) {
18             vector_current[i]=vector_i[i];
19         }
20
21         Load_bias_loop: for (int j=0; j< BIAS_SIZE_VECTOR; j++) {
22             bias_current[j]=bias_i[j];
23         }
24     }
25     else {
26         if (*wr==1) {
```

```
27              Write_ewop_loop: for (int i=0; i<GATE_SIZE_VECTOR*2; i++)
    {
28                  vector_o[i+(ARRAY_SIZE_VECTOR-2*GATE_SIZE_VECTOR)]=
    ewop_o[i];
29              }
30          }
31          if (*rd==1) { //vector output
32              Out_loop_v: for (int i=0; i<ARRAY_SIZE_VECTOR; i++) {
33              vector_o[i]=vector_current[i];
34              }
35              Out_loop_b: for (int i=0; i<BIAS_SIZE_VECTOR; i++) {
36                  bias_o[i]=bias_current[i];
37              }
38          }
39      }
40  }
```

A.4 PE Unit

A.4.1 Header File

```
1
2  #ifndef _PE_H_
3  #define _PE_H_
4
5  #include <stdio.h>
6
7  #define NON_ZERO_ROW 8  //non-zero values in one row in pruned matrix
8  #define NUM_BANK 4  //N vector elements into private vector buffer
       from vector memory
9  #define BANK_SIZE 4 //the size of each bank
10
11 #define MATRIX_ROW 16
12
13 //adder tree
14 #define VALUE_SIZE_TREE 4 //the size of value of matrix from matrix
       memory
15 #define VECTOR_SIZE_TREE 4
16 #define ACCUM_LEN 10
17
18 //PE
19 #define VALUE_SIZE_PE  4
20 #define INDX_SIZE_PE   4     //the size of bank internal indices from
       matrix memory
21 #define VECTOR_SIZE_PE 16    //vector and value from memory
```

```
22
23 typedef float din_t;
24 typedef float dout_t;
25 typedef int dindx_t;
26 typedef int cin_t;
27
28 dout_t pe(din_t vector_i[VECTOR_SIZE_PE], din_t value_i[VALUE_SIZE_PE
       ], dindx_t vector_idx[INDX_SIZE_PE]);
29
30 #endif
```

A.4.2 PE.cpp

```
1
2 #include "PE.h"
3
4 //private vector buffer
5 //split vector array into banks corresponding to banks in pruned
      matrix row
6 //here split into 4 banks, and pick one element into pvb_vector_o
7
8 void pvb (dout_t pvb_vector_o[NUM_BANK], din_t vector_i[NUM_BANK*
      BANK_SIZE], dindx_t vector_idx[INDX_SIZE_PE]) {
9     int i=0;
10    int j=0;
11    int k=0;  //count for output vector element from each bank
12    dout_t vector_bank[BANK_SIZE];
13
14
15    Fill_all_bank_loop: for (i=0; i<(NUM_BANK*BANK_SIZE); i=i+
      BANK_SIZE) {
16            Fill_one_bank_loop: for (j=0; j<BANK_SIZE; j++) {
17
18                vector_bank[j]=vector_i[i+j];
19            }
20
21            pvb_vector_o[k]=vector_bank[vector_idx[k]];
22            k=k+1;
23        }
24 }
25
26 //adder tree-generic
27 dout_t adder_tree (din_t value_i[VALUE_SIZE_TREE], din_t vector_i[
      VECTOR_SIZE_TREE]) {
28    int i;
29    int j;
```

```
30      static dout_t accum[ACCUM_LEN]; //array for accumulation
31      dout_t accum_final;
32
33      add_binary_loop: for (i=0; i<(VALUE_SIZE_TREE/2); i++) {
34          accum[i]=value_i[2*i]*vector_i[2*i]+value_i[2*i+1]*vector_i
        [2*i+1];
35      }
36
37      add_accum_loop: for(j=0; j<(VALUE_SIZE_PE/2); j++) {
38          accum_final=accum_final+accum[j];
39      }
40
41      return accum_final;
42  }
43
44
45  dout_t pe(din_t vector_i[VECTOR_SIZE_PE], din_t value_i[VALUE_SIZE_PE
        ], dindx_t vector_idx[INDX_SIZE_PE]) {
46      int i;
47      static dout_t pvb_vector_o[NUM_BANK];  //pointer of ouput of pvb
48      dout_t pe_o;
49
50      pvb(pvb_vector_o, vector_i, vector_idx);  //output of pvb, into
        adder tree
51      pe_o=adder_tree(value_i, pvb_vector_o);
52
53      return pe_o;
54  }
```

A.5 SpMxV Unit

A.5.1 Header file

```
1
2   #ifndef _SPMXV_H_
3   #define _SPMXV_H_
4
5   #include <stdio.h>
6
7   #define _CRT_SECURE_NO_WARNINGS
8
9   #define NUM_PE 2
10
11  //assume the weight matrix is 16*16
12  //split 4 bank in each row, i.e., 1*4 bank
```

```
13 //#of non-zero value in each bank is 2 (50% purning rate)
14 //therefore 8 non-zero value in each row, total 8*16=128 non-zero
       value in the matrix
15
16 #define VALUE_SIZE  128
17 #define INDX_SIZE   128
18 #define VECTOR_SIZE 16
19 #define BIAS_SIZE   16
20
21 #define BANK_SIZE   4
22 #define NUM_BANK    4
23 #define NNZ_BANK    2
24
25 #define OUTSIZE     16
26
27 typedef float din_t;
28 typedef float dout_t;
29
30 typedef int dindx_t;
31
32 typedef int cin_t;
33
34 void spmxv(dout_t result_vector[OUTSIZE], din_t value_in[VALUE_SIZE],
       din_t vector_in[VECTOR_SIZE], dindx_t vector_idx[INDX_SIZE],
      din_t bias_i[BIAS_SIZE]);
35
36 #endif
```

A.5.2 SpMxV.cpp

```
1
2 #include "SpMxV.h"
3 #include "PE.h"
4
5 //spmxv unit, value_in and vector_in are from matrix memory and
      vector memory respectively
6 //ins_read and ins_write are the enable signals to read and write
      respectively, from instruction buffer
7
8 void spmxv(dout_t result_vector[OUTSIZE], din_t value_in[VALUE_SIZE],
       din_t vector_in[VECTOR_SIZE], dindx_t vector_idx[INDX_SIZE],
      din_t bias_i[BIAS_SIZE]) {
9
10     dout_t pe_o;
11     int i;
12
```

```
13      static din_t value[NUM_BANK];
14      static dindx_t indx[NUM_BANK];
15      int count=0;
16
17      //clear result_vector
18      clean_loop: for(int m=0; m<OUTSIZE; m++) {
19              result_vector[m]=0;
20          }
21
22      Matrix_row_loop: for (int k=0; k<VALUE_SIZE; k=k+NUM_BANK*NUM_PE)
        {
23          PE_LOOP: for (i=0; i<NUM_PE*NUM_BANK; i=i+NUM_BANK) {
24              Bank_gen_loop: for (int j=0; j<NUM_BANK; j++) {
25                      value[j]=value_in[i+j+k];
26                      indx[j]=vector_idx[i+j+k];
27                  }
28                  pe_o=pe(vector_in, value, indx);
29                  result_vector[count]=result_vector[count]+pe_o;
30          }
31                  result_vector[count]=result_vector[count]+bias_i[
    count];
32                  count++;
33      }
34
35  }
```

A.6 EWOP Unit

A.6.1 Header file

```
1
2  #ifndef _EWOP_H_
3  #define _EWOP_H_
4  #include "cmath"
5
6  #define GATE_SIZE 4
7  #define LOAD_MEM  3
8
9
10 typedef float din_t;
11 typedef float dout_t;
12 typedef int dindx_t;
13 typedef int cin_t;
14 typedef int cout_t;
15
```

```
16 void ewop(dout_t ewop_o[GATE_SIZE*2], cout_t *en_wr_v, din_t spmxv_o[
       GATE_SIZE*4], cin_t ir_in);
17 dout_t sigmoid(din_t x);
18 dout_t tanh(din_t x);
19
20 #endif
```

A.6.2 EWOP.cpp

```
1
2 #include "EWOP.h"
3
4 void ewop(dout_t ewop_o[GATE_SIZE*2], cout_t *en_wr_v, din_t spmxv_o[
       GATE_SIZE*4], cin_t ir_in) {
5
6     static din_t gate[GATE_SIZE*4];
7     static din_t i_gate[GATE_SIZE];
8     static din_t f_gate[GATE_SIZE];
9     static din_t g_gate[GATE_SIZE];
10    static din_t o_gate[GATE_SIZE];
11
12    static dout_t c_state[GATE_SIZE];
13    static dout_t y[GATE_SIZE];
14
15    //clear the state
16    clear_loop: for (int k=0; k<GATE_SIZE; k++) {
17            c_state[k]=0;
18            y[k]=0;
19        }
20
21 //pipeline
22    Gate_value_loop: for (int i=0; i<GATE_SIZE*4; i++) {
23        gate[i]=sigmoid(spmxv_o[i]);  //gate value
24    }
25
26 //split the gate vectors
27    Split_gate_loop: for (int j=0; j<GATE_SIZE*4; j++) {
28        if(j<GATE_SIZE) {
29            i_gate[j]=gate[j];
30        }
31        else if (j<GATE_SIZE*2) {
32            g_gate[j-GATE_SIZE]=gate[j];
33        }
34        else if (j<GATE_SIZE*3) {
35            f_gate[j-2*GATE_SIZE]=gate[j];
36        }
```

```
37          else {
38              o_gate[j-3*GATE_SIZE]=gate[j];
39          }
40      }
41
42      computation_loop: for (int k=0; k<GATE_SIZE; k++) {
43          c_state[k]=c_state[k]*f_gate[k]+g_gate[k]*i_gate[k];
44          y[k]=o_gate[k]*tanh(c_state[k]);
45      }
46
47  //generate ewop_o output to vector memory to create new vector
48      Ewop_gen_loop: for (int m=0; m<GATE_SIZE; m++) {
49          ewop_o[m]=y[m];
50          ewop_o[m+GATE_SIZE]=c_state[m];
51      }
52
53      if(ir_in!=LOAD_MEM) {
54      *en_wr_v=1;
55      }
56
57  }
```

A.7 Activation Functions

A.7.1 Header File

```
1
2  #ifndef __ACTIVEMODULE_H__
3  #define __ACTIVEMODULE_H__
4
5  #include "cmath"
6
7  typedef float din_t;
8  typedef float dout_t;
9
10
11 dout_t SigmoidFunc(din_t x);
12 dout_t TanhFunc(din_t x);
13
14 #endif
```

A.7.2 Sigmoid.cpp

```
1
2 #include "ActiveModule.h"
3
4 #define N_slice 160
5
6 dout_t sigmoid(din_t x)
7
8 {
9     static float sigmoid_a[N_slice]= {
10        0.000352560101599184,    0.000389610243596535,
    0.000430550582544846,    0.000475788838390289,
11        0.000525775298440352,    0.000581007225108015,
    0.000642033711984364,    0.000709461032038589,
12        0.000783958525615976,    0.000866265080023501,
    0.000957196256849915,    0.00105765212774684,
13        0.00116862588417099,     0.00129121329151538,
    0.00142662306308033,     0.00157618823437430,
14        0.00174137862318382,     0.00192381446557077,
    0.00212528132226450,     0.00234774635358441,
15        0.00259337606376285,     0.00286455561697362,
    0.00316390982704920,     0.00349432592022642,
16        0.00385897816460139,     0.00426135445045051,
    0.00470528489115779,     0.00519497249393442,
17        0.00573502592133708,     0.00633049432700738,
    0.00698690419997115,     0.00771029808877919,
18        0.00850727499687927,     0.00938503214056493,
    0.0103514076368886,      0.0114149235368106,
19        0.0125848284339053, 0.0138711386560828, 0.0152846767816788,
    0.0168371059065062,
20        0.0185409577198595, 0.0204096520205297, 0.0224575048153873,
    0.0246997215919666,
21        0.0271523717449047, 0.0298323394709420, 0.0327572457418567,
    0.0359453352412819,
22        0.0394153214432177, 0.0431861823648066, 0.0472768990085096,
    0.0517061282045102,
23        0.0564918015812776, 0.0616506428635033, 0.0671975967789674,
    0.0731451647267883,
24        0.0795026452093378, 0.0862752810482902, 0.0934633207592782,
    0.101061008265046,
25        0.109055523408803,  0.117425905374900,  0.126142001830469,
    0.135163497825408,
26        0.144439089402808,  0.153905876350619,  0.163489055160232,
    0.173101995435410,
27        0.182646779039001,  0.192015269651127,  0.201090760050009,
    0.209750214973916,
28        0.217867089594464,  0.225314659423707,  0.231969750239409,
    0.237716710894026,
```

```
29      0.242451433007930,  0.246085194991811,  0.248548098335379,
    0.249791874789400,
30      0.249791874789400,  0.248548098335379,  0.246085194991811,
    0.242451433007930,
31      0.237716710894026,  0.231969750239408,  0.225314659423707,
    0.217867089594462,
32      0.209750214973915,  0.201090760050010,  0.192015269651128,
    0.182646779039001,
33      0.173101995435410,  0.163489055160232,  0.153905876350620,
    0.144439089402808,
34      0.135163497825409,  0.126142001830468,  0.117425905374901,
    0.109055523408804,
35      0.101061008265048,  0.0934633207592772, 0.0862752810482892,
    0.0795026452093373,
36      0.0731451647267889, 0.0671975967789673, 0.0616506428635022,
    0.0564918015812776,
37      0.0517061282045106, 0.0472768990085104, 0.0431861823648050,
    0.0394153214432180,
38      0.0359453352412820, 0.0327572457418568, 0.0298323394709410,
    0.0271523717449040,
39      0.0246997215919653, 0.0224575048153863, 0.0204096520205310,
    0.0185409577198592,
40      0.0168371059065064, 0.0152846767816783, 0.0138711386560819,
    0.0125848284339058,
41      0.0114149235368111, 0.0103514076368894, 0.00938503214056308,
        0.00850727499688064,
42      0.00771029808877999,    0.00698690419997150,
    0.00633049432700483,    0.00573502592133734,
43      0.00519497249393486,    0.00470528489115885,
    0.00426135445045106,    0.00385897816460146,
44      0.00349432592022581,    0.00316390982704973,
    0.00286455561697374,    0.00259337606376309,
45      0.00234774635358259,    0.00212528132226497,
    0.00192381446557088,    0.00174137862318347,
46      0.00157618823437455,    0.00142662306308106,
    0.00129121329151549,    0.00116862588417144,
47      0.00105765212774611,    0.000957196256849979,
    0.000866265080022854,   0.000783958525616413,
48      0.000709461032039060,   0.000642033711983903,
    0.000581007225107788,   0.000525775298441111,
49      0.000475788838389946,   0.000430550582544154,
    0.000389610243596295,   0.000352560101600785
50  };
51
52  static float sigmoid_b[N_slice]= {
53      0.00315583094325995, 0.00344852706503903,
    0.00376786170883585,    0.00411619627884576,
54      0.00449609337522624, 0.00491033282523371,
    0.00536192882811870,    0.00585414826451454,
```

```
55          0.00639053021827172, 0.00697490675456515,
    0.00761142499235005,     0.00830457050153880,
56          0.00905919204522303, 0.00988052767443046,
    0.0107742321667591, 0.0117464057801699,
57          0.0128036242685509, 0.0139529700755886, 0.0152020645870898,
    0.0165591012781412,
58          0.0180328795392119, 0.0196328389031554, 0.0213690933215938,
    0.0232524650527039,
59          0.0252945176212038, 0.0275075871933739, 0.0299048115731932,
    0.0325001558679094,
60          0.0353084336904032, 0.0383453225593218, 0.0416273719241406,
    0.0451720019793000,
61          0.0489974911381804, 0.0531229497135030, 0.0575682769965917,
    0.0623540985462408,
62          0.0675016800934576, 0.0730328140488207, 0.0789696741763240,
    0.0853346335881164,
63          0.0921500408415296, 0.0994379486141434, 0.107219789234602,
    0.115515991307946,
64          0.124345531858523,  0.133725418899653,  0.143670100220763,
    0.154190795568866,
65          0.165294751415061,  0.176984420271987,  0.189256570203095,
    0.202101334871497,
66          0.215501220326446,  0.229430091788455,  0.243852171968662,
    0.258721091838214,
67          0.273979044996333,  0.289556107425924,  0.305369794790097,
    0.321324938552210,
68          0.337313968839724,  0.353217694575307,  0.368906668195333,
    0.384243211386729,
69          0.399084157910569,  0.413284338332285,  0.426700788665743,
    0.439197611023474,
70          0.450651351347784,  0.460956691021122,  0.470032181420004,
    0.477825690851520,
71          0.484319190547959,  0.489532489428429,  0.493525543917850,
    0.496399024245158,
72          0.498292913090720,  0.499383041685884,  0.499875622354598,
    0.500000000000000,
73          0.500000000000000,  0.500124377645402,  0.500616958314116,
    0.501707086909280,
74          0.503600975754842,  0.506474456082151,  0.510467510571572,
    0.515680809452043,
75          0.522174309148481,  0.529967818579995,  0.539043308978877,
    0.549348648652217,
76          0.560802388976526,  0.573299211334257,  0.586715661667714,
    0.600915842089432,
77          0.615756788613270,  0.631093331804669,  0.646782305424691,
    0.662686031160274,
78          0.678675061447787,  0.694630205209905,  0.710443892574079,
    0.726020955003668,
```

```
79          0.741278908161784,  0.756147828031338,  0.770569908211548,
    0.784498779673554,
80          0.797898665128502,  0.810743429796902,  0.823015579728018,
    0.834705248584938,
81          0.845809204431133,  0.856329899779237,  0.866274581100350,
    0.875654468141480,
82          0.884484008692059,  0.892780210765401,  0.900562051385852,
    0.907849959158472,
83          0.914665366411883,  0.921030325823678,  0.926967185951183,
    0.932498319906540,
84          0.937645901453757,  0.942431723003405,  0.946877050286506,
    0.951002508861813,
85          0.954827998020696,  0.958372628075858,  0.961654677440691,
    0.964691566309595,
86          0.967499844132088,  0.970095188426801,  0.972492412806623,
    0.974705482378796,
87          0.976747534947300,  0.978630906678403,  0.980367161096844,
    0.981967120460787,
88          0.983440898721870,  0.984797935412907,  0.986047029924411,
    0.987196375731451,
89          0.988253594219828,  0.989225767833236,  0.990119472325569,
    0.990940807954774,
90          0.991695429498466,  0.992388575007650,  0.993025093245439,
    0.993609469781725,
91          0.994145851735482,  0.994638071171885,  0.995089667174768,
    0.995503906624768,
92          0.995883803721157,  0.996232138291170,  0.996551472934963,
    0.996844169056727
93      };
94
95      dout_t y;
96      dout_t a;
97      dout_t b;
98      int index;
99
100     if (x<=-8.0) {
101         y=0;
102     } else {
103         if ((x>-8.0) && (x<=0)) {
104             index=int((x+8.0)/0.1);
105             a=sigmoid_a[index];
106             b=sigmoid_b[int((x+8.0)/0.1)];
107             y=a*x+b;
108         } else {
109             if ((x>0) && (x<8.0)) {
110                 y=sigmoid_a[int((x+8.0)/0.1)]*x+sigmoid_b[int((x+8.0)
    /0.1)];
111             } else {
112                 y=1.0;
```

```
113                 }
114             }
115         }
116 
117         return y;
118 
119 }
```

A.7.3 Tanh.cpp

```
1 
2 #include "ActiveModule.h"
3 
4 
5 #define N_slice 120
6 
7 dout_t tanh(din_t x)
8 
9 {
10     static float tanh_a[N_slice]= {
11         0.0000272065398976284,  0.0000332300424554788,
    0.000040587115704982,   0.0000495729915894749,
12         0.0000605482552706871,  0.0000739533086357991,
    0.0000903260331908129,  0.000110323359222564,
13         0.000134747604045860,   0.000164578631991485,
    0.000201013120406657,   0.000245512497939471,
14         0.000299861464638784,   0.000366239421121239,
    0.000447307641570083,   0.000546315641469786,
15         0.000667230938073971,   0.000814897305396745,
    0.000995227717360825,   0.00121543948730318,
16         0.00148434069039083,    0.00181267884187042,
    0.00221356504709669,    0.00270298948804593,
17         0.00330044721127964,    0.00402969676919329,
    0.00491967835137275,    0.006005622594909,
18         0.00733038617750981,    0.0089460553516973 5,
    0.0109158633614737, 0.0133164714945511,
19         0.0162406652301039, 0.0198005147701841, 0.0241310404966888,
    0.0293944045775008,
20         0.0357846142748886, 0.0435326623473986, 0.0529119341799766,
    0.0642435653763696,
21         0.0779012194807782, 0.0943144528147077, 0.113969422431693,
    0.137405161966276,
22         0.165203007616048,  0.197966054426039,  0.236284888889562,
    0.280685523011511,
23         0.331555852515257,  0.389048658048649,  0.452962857567404,
    0.522610999311755,
```

```
24      0.596689931506855,  0.673182101191282,  0.749324097380256,
    0.821681950047849,
25      0.886363498036340,  0.939372922266868,  0.977073255999483,
    0.996679946249558,
26      0.996679946249559,  0.977073255999481,  0.939372922266868,
    0.886363498036338,
27      0.821681950047848,  0.749324097380253,  0.673182101191283,
    0.596689931506855,
28      0.522610999311757,  0.452962857567405,  0.38904865804865,
    0.331555852515253,
29      0.280685523011512,  0.236284888889560,  0.197966054426042,
    0.165203007616046,
30      0.137405161966278,  0.113969422431690,  0.0943144528147033,
    0.0779012194807804,
31      0.0642435653763696, 0.0529119341799755, 0.0435326623474008,
    0.0357846142748874,
32      0.0293944045775030, 0.0241310404966844, 0.0198005147701874,
    0.016240665230105,
33      0.0133164714945511, 0.0109158633614737, 0.00894605535169513,
        0.0073303861775087,
34      0.00600562259491122,    0.00491967835137386,
    0.00402969676919218,    0.00330044721127853,
35      0.00270298948804593,    0.00221356504709780,
    0.00181267884186820,    0.00148434069039416,
36      0.00121543948729874,    0.000995227717361935,
    0.000814897305398965,   0.000667230938073971,
37      0.000546315641467565,   0.000447307641571193,
    0.000366239421123460,   0.000299861464638784,
38      0.00024551249793614,    0.000201013120408877,
    0.000164578631989265,   0.00013474760404586,
39      0.000110323359225895,   0.0000903260331885925,
    0.0000739533086391297,  0.0000605482552673564,
40      0.0000495729915916954,  0.0000405871157038717,
    0.0000332300424532583,  0.000027206539898738 7
41      };
42
43  static float tanh_b[N_slice]= {
44      -0.999824472411410, -0.999788933746319, -0.999746262721471,
    -0.99969504322893,
45      -0.999633581752315, -0.999559853958807, -0.999471441246210,
    -0.999365455418242,
46      -0.999238449345160, -0.999086311102638, -0.998904138660562,
    -0.998686091710651,
47      -0.998425216670494, -0.998113240275027, -0.997740326460962,
    -0.997294790461413,
48      -0.996762763156355, -0.996127797776867, -0.995370410046618,
    -0.994467541789854,
49      -0.993391936977504, -0.992111418186733, -0.990588050606874,
    -0.988777180175361,
```

```
50          -0.986626332371720, -0.984073958919022, -0.981048021539612,
    -0.977464405535942,
51          -0.973225162071620, -0.968216587631638, -0.962307163602310,
    -0.955345400016385,
52          -0.947157657556837, -0.937546063798621, -0.926286696909708,
    -0.913128286707678,
53          -0.897791783433948, -0.879971272867175, -0.859336874835503,
    -0.835540449323078,
54          -0.808225141114261, -0.777039997779794, -0.74166105246922,
    -0.701820295260429,
55          -0.657343742220794, -0.608199172005808, -0.554552803756876,
    -0.496831979398342,
56          -0.435787583993847, -0.372545497907116, -0.308631298388361,
    -0.245947970818445,
57          -0.186684825062365, -0.133140306283266, -0.0874551085698817,
        -0.0512761822360853,
58          -0.025403563040689, -0.00950073577153049,
    -0.00196066902500738,    0,
59          0,                    0.00196066902500772,
    0.00950073577153049,    0.0254035630406897,
60          0.0512761822360858, 0.0874551085698834, 0.133140306283266,
    0.186684825062365,
61          0.245947970818443,  0.30863129838836,   0.372545497907115,
    0.435787583993852,
62          0.496831979398340,  0.554552803756879,  0.608199172005804,
    0.657343742220798,
63          0.701820295260427,  0.741661052469226,  0.777039997779803,
    0.808225141114256,
64          0.835540449323078,  0.859336874835505,  0.87997127286717,
    0.89779178343395,
65          0.913128286707673,  0.926286696909720,  0.937546063798612,
    0.947157657556834,
66          0.955345400016385,  0.962307163602310,  0.968216587631645,
    0.973225162071623,
67          0.977464405535935,  0.981048021539609,  0.984073958919026,
    0.986626332371724,
68          0.988777180175361,  0.990588050606869,  0.992111418186742,
    0.993391936977491,
69          0.994467541789872,  0.995370410046613,  0.996127797776858,
    0.996762763156355,
70          0.997294790461423,  0.997740326460957,  0.998113240275017,
    0.998425216670495,
71          0.998686091710667,  0.998904138660551,  0.999086311102649,
    0.99923844934516,
72          0.999365455418224,  0.999471441246222,  0.999559853958789,
    0.999633581752334,
73          0.999695043228917,  0.999746262721478,  0.999788933746332,
    0.999824472411403
74          };
```

```
75
76          dout_t y;
77
78      if (x<=-6.0) {
79          y=-1.0;
80      } else {
81          if ((x>-6.0) && (x<=0)) {
82              y=tanh_a[int((x+6.0)/0.1)]*x-tanh_b[int((x+6.0)/0.1)];
83          } else {
84              if ((x>0) && (x<6.0)) {
85                  y=tanh_a[int((x+6.0)/0.1)]*x+tanh_b[int((x+6.0)/0.1)
    ];
86              } else {
87                  y=1.0;
88              }
89          }
90      }
91      return y;
92  }
```

A.8 Control Unit

A.8.1 Header File

```
1
2  #ifndef _ControlUnit_H_
3  #define _ControlUnit_H_
4
5  #include "stdio.h"
6
7  #define READ_PARA     1    //read parameters: weight matrix and vectors
8  #define NO_READ       2
9  #define LOAD_MEM      3
10
11 //data types of control signals: in & out
12 typedef int cin_t;
13 typedef int cout_t;
14
15 void cu(cout_t *rd_matrix, cout_t *rd_vector, cout_t *wr_vector,
       cout_t *ld_matrix, cout_t *ld_vector, cin_t ir_in, cin_t *en_wr_v)
       ;
16
17 #endif
```

A.8.2 ControlUnit.cpp

```
1
2 #include "ControlUnit.h"
3
4 /*control singals:
5 in:
6 clk
7 rst
8 ir_in: the instruction
9
10 out:
11 *matrix memory:
12 rd_matrix: read csb_value and csb vectors into spmxv from matrix
       memory
13 *vector memory:
14 rd_vector: read vector into spmxv from vector memory
15 wr_vector: write y from ewop into vector memory to form new vector
16
17 */
18
19 void cu(cout_t *rd_matrix, cout_t *rd_vector, cout_t *wr_vector,
       cout_t *ld_matrix, cout_t *ld_vector, cin_t ir_in, cin_t *en_wr_v)
        {
20     if (ir_in==LOAD_MEM) {    //asynchronous reset (active low)
21         *rd_matrix=0;
22         *rd_vector=0;
23         *wr_vector=0;
24         *ld_matrix=1;
25         *ld_vector=1;
26     }
27     else if (ir_in==READ_PARA) {
28               *ld_matrix=0;
29               *ld_vector=0;
30               *rd_matrix=1;
31               *rd_vector=1;
32     }
33     else if(ir_in==NO_READ) {
34              *ld_matrix=0;
35              *ld_vector=0;
36              *rd_matrix=0;
37              *rd_vector=0;
38     }
39
40     if(*en_wr_v==1) {     //status signal: enable writing output of
       ewop unit in vector memory to create new vector
41              *wr_vector=1;
```

```
42      }
43  }
```

A.9 Testbench

```
1
2  #include "BBS.h"
3
4      cin_t ir;
5
6  int main() {
7
8
9  //input data
10     din_t values[MEM_SIZE_M];
11     dindx_t index[MEM_SIZE_M];
12     din_t vector[MEM_SIZE_V];
13     din_t bias[MEM_SIZE_B];
14
15 //output
16     dout_t ewop_o[GATE_SIZE*2];
17
18 //golden data of output
19     dout_t ewop_golden[GATE_SIZE*2];
20
21 //misc
22         FILE *fp;
23         int retval=0;
24
25
26 //load input data from files
27         fp=fopen("csb_values.dat", "r"); //open data files
28         for (int i=0; i< VALUE_SIZE; i++) {
29             din_t tmp;
30             fscanf(fp, "%f", &tmp);
31             values[i]=tmp;
32         }
33         fclose(fp);
34
35
36         fp=fopen("csb_index.dat", "r");
37         for (int i=0; i< INDX_SIZE; i++) {
38             dindx_t tmp;
39             fscanf(fp, "%d", &tmp);
40             index[i]=tmp;
```

```
41          }
42          fclose(fp);
43
44          fp=fopen("vector.dat", "r");
45          for (int i=0; i< VECTOR_SIZE; i++) {
46              din_t tmp;
47              fscanf(fp, "%f", &tmp);
48              vector[i]=tmp;
49          }
50          fclose(fp);
51
52
53          fp=fopen("bias.dat", "r");
54          for (int i=0; i< BIAS_SIZE; i++) {
55              din_t tmp;
56              fscanf(fp, "%f", &tmp);
57              bias[i]=tmp;
58          }
59          fclose(fp);
60
61  //load expected output data from file
62          fp=fopen("ewop_golden.dat", "r");
63          for (int i=0; i< GATE_SIZE*2; i++) {
64              din_t tmp;
65              fscanf(fp, "%f", &tmp);
66              ewop_golden[i]=tmp;
67          }
68          fclose(fp);
69
70  //reset
71      printf("start reseting\n");
72      ir=LOAD_MEM;
73      bbs_arch(ewop_o, values, index, vector, bias, ir);
74      printf("finish reseting\n");
75
76  //run
77      printf("start running\n");
78      ir=READ_PARA;
79      bbs_arch(ewop_o, values, index, vector, bias, ir);
80      printf("finishing running\n");
81
82
83  //check outputs agianst expected, the threshold is 0.01
84      for (int i=0; i<GATE_SIZE*2; i++) {
85          printf("ewop %d is %f, ewop golden %d is %f\n", i, ewop_o[i],
        i, ewop_golden[i]);
86          if (fabs(ewop_golden[i]-ewop_o[i])>0.1) {
87              retval=1;
88          }
```

```
89      }
90
91      //print results
92          if(retval ==0) {
93              printf("*****************\n");
94              printf("RESULTS ARE GOOD!\n");
95              printf("*****************\n");
96          } else {
97              printf("*****************\n");
98              printf("Mismatch:retval=%d \n", retval);
99              printf("*****************\n");
100         }
101
102 //return 0 if output are correct
103         return retval;
104
105 }
```

Appendix B

Scripts

B.1 run_hls.tcl

```
1
2 # Project settings
3
4 # Create a project
5 open_project    -reset bbs_prj
6
7 # The source file and test bench
8
9 #SPMXV UNIT
10 add_files            SpMxV.cpp
11 add_files            SpMxV.h
12 add_files            PE.cpp
13 add_files            PE.h
14
15 #EWOP UNIT
16 add_files            SigmoidFunc.cpp
17 add_files            TanhFunc.cpp
18 add_files            EWOP.cpp
19 add_files            EWOP.h
20
21 #MEM
22 add_files            MatrixMem.cpp
23 add_files            MatrixMem.h
24 add_files            VectorMem.cpp
25 add_files            VectorMem.h
26
27 #Control Unit
28 add_files            ControlUnit.cpp
29 add_files            ControlUnit.h
```

```
30
31 #Top level
32 add_files            BBS.cpp
33 add_files            BBS.h
34
35 add_files -tb    TB_BBS.cpp
36 # Specify the top-level function for synthesis
37 set_top              bbs_arch
38
39
40 # Solution settings
41
42 # Create solution1
43 open_solution -reset solution1
44
45 # Specify a Xilinx device and clock period
46 # - Do not specify a clock uncertainty (margin)
47 # - Let the  margin to default to 12.5% of clock period
48 set_part  {xc7z020clg400-1}
49 create_clock -period 8
50 #set_clock_uncertainty 1.25
51
52 # Simulate the C code
53 csim_design
54
55 # Do not perform any other steps
56 # - The basic project will be opened in the GUI
57 exit
```

Appendix C

Software Drivers

```
1
2  #include <stdio.h>
3  #include "platform.h"
4  #include "xil_printf.h"
5  #include "xparameters.h"
6  #include "xil_printf.h"
7  #include "sleep.h"
8  #include "xil_cache.h"
9  #include "xbbs_arch.h"
10
11 float ewop_o[8];
12
13 //inputs
14 float value_in[128] = {0.919172,
       -0.558193,0.777594,0.515993,-0.069569..};
15 int index_in[128]={1, 0, 0, 0,2...};
16 float vector_in[16]={-0.3414,2.85878, 2.03407,..};
17 float bias_in[16]={ 3.66618,2.74954,2.63324, 0.947678,..};
18 float ewop_o_soft[8]={ 0.085166, 0.567416, 0.022210, 0.355274..}
19
20 int ir_in;
21
22 void RunBbs(XBbs_arch *InstancePtr, float ewop_o[], float value_in[],
       int index_in[], float vector_in[], float bias_in[], int ir_in)
23
24 {
25     XBbs_arch_Set_ewop_o(InstancePtr,(unsigned int)ewop_o);
26     XBbs_arch_Set_value_i(InstancePtr,(unsigned int)value_in);
27     XBbs_arch_Set_index_i(InstancePtr,(unsigned int)index_in);
28     XBbs_arch_Set_vector_i(InstancePtr,(unsigned int)vector_in);
29     XBbs_arch_Set_bias_i(InstancePtr,(unsigned int)bias_in);
```

```
30      XBbs_arch_Set_ir_in(InstancePtr, ir_in);
31
32
33
34      XBbs_arch_Start(InstancePtr);
35      while(!XBbs_arch_IsDone(InstancePtr));
36  }
37
38  int main()
39  {
40      init_platform();
41      Xil_DCacheDisable();
42      char str[50];
43
44      print("Hello\n\r");
45
46      XBbs_arch bbs_arch;
47      if(XBbs_arch_Initialize(&bbs_arch, XPAR_BBS_ARCH_0_DEVICE_ID)!=
        XST_SUCCESS) {
48          xil_printf("XBbs device is not found\n\r");
49      }
50
51  //preset
52      ir_in=3;  //load
53      RunBbs(&bbs_arch, ewop_o, value_in, index_in, vector_in, bias_in,
         ir_in);
54      for (int i=0; i<8; i++) {
55          sprintf(str,"ewop[%d]=%f\r\n",i,ewop_o[i]);
56          xil_printf(str);
57          //sprintf(str,"ewop_soft[%d]=%f\r\n",i,ewop_o_soft[i]);
58          //xil_printf(str);
59        }
60
61  //run
62      ir_in=1;  //run
63      RunBbs(&bbs_arch, ewop_o, value_in, index_in, vector_in, bias_in,
         ir_in);
64      for (int i=0; i<8; i++) {
65          sprintf(str,"ewop[%d]=%f, ewop_golden[%d]=%f\r\n",i,ewop_o[i
        ],i,ewop_o_soft[i]);
66          xil_printf(str);
67      }
68
69      printf("end\n");
70
71      cleanup_platform();
72      return 0;
73  }
```

Bibliography

[1] Jürgen Schmidhuber. «Deep learning in neural networks: An overview». In: *Neural networks* 61 (2015), pp. 85–117 (cit. on pp. 1, 5).

[2] Alex Graves, Abdel-rahman Mohamed, and Geoffrey Hinton. «Speech recognition with deep recurrent neural networks». In: *2013 IEEE international conference on acoustics, speech and signal processing*. IEEE. 2013, pp. 6645–6649 (cit. on p. 1).

[3] Yonghui Wu et al. «Google's neural machine translation system: Bridging the gap between human and machine translation». In: *arXiv preprint arXiv:1609.08144* (2016) (cit. on p. 1).

[4] Sepp Hochreiter. «The vanishing gradient problem during learning recurrent neural nets and problem solutions». In: *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems* 6.02 (1998), pp. 107–116 (cit. on pp. 1, 5, 10).

[5] Sepp Hochreiter and Jürgen Schmidhuber. «Long short-term memory». In: *Neural computation* 9.8 (1997), pp. 1735–1780 (cit. on pp. 1, 5, 10).

[6] Stephen D Brown, Robert J Francis, Jonathan Rose, and Zvonko G Vranesic. *Field-programmable gate arrays*. Vol. 180. Springer Science & Business Media, 2012 (cit. on p. 1).

[7] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. «Optimizing fpga-based accelerator design for deep convolutional neural networks». In: *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 2015, pp. 161–170 (cit. on p. 2).

[8] S Himavathi, D Anitha, and A Muthuramalingam. «Feedforward neural network implementation in FPGA using layer multiplexing for effective resource utilization». In: *IEEE Transactions on Neural Networks* 18.3 (2007), pp. 880–888 (cit. on p. 2).

[9] Yiwei Zhang, Chao Wang, Lei Gong, Yuntao Lu, Fan Sun, Chongchong Xu, Xi Li, and Xuehai Zhou. «Implementation and optimization of the accelerator based on fpga hardware for lstm network». In: *2017 IEEE International Symposium on Parallel and Distributed Processing with Applications and 2017 IEEE International Conference on Ubiquitous Computing and Communications (ISPA/IUCC)*. IEEE. 2017, pp. 614–621 (cit. on pp. 2, 13, 22, 25, 47).

[10] Song Han, Jeff Pool, John Tran, and William Dally. «Learning both weights and connections for efficient neural network». In: *Advances in neural information processing systems*. 2015, pp. 1135–1143 (cit. on p. 2).

[11] Huizi Mao, Song Han, Jeff Pool, Wenshuo Li, Xingyu Liu, Yu Wang, and William J Dally. «Exploring the regularity of sparse structure in convolutional neural networks». In: *arXiv preprint arXiv:1705.08922* (2017) (cit. on pp. 2, 14).

[12] Runbin Shi, Junjie Liu, K-H Hayden So, Shuo Wang, and Yun Liang. «E-LSTM: Efficient Inference of Sparse LSTM on Embedded Heterogeneous System». In: *2019 56th ACM/IEEE Design Automation Conference (DAC)*. IEEE. 2019, pp. 1–6 (cit. on pp. 2, 23).

[13] Jaeha Kung, Junki Park, Sehun Park, and Jae-Joon Kim. «Peregrine: A flexible hardware accelerator for LSTM with limited synaptic connection patterns». In: *Proceedings of the 56th Annual Design Automation Conference 2019*. 2019, pp. 1–6 (cit. on pp. 2, 23).

[14] Sharan Narang, Eric Undersander, and Gregory Diamos. «Block-sparse recurrent neural networks». In: *arXiv preprint arXiv:1711.02782* (2017) (cit. on p. 2).

[15] Shijie Cao, Chen Zhang, Zhuliang Yao, Wencong Xiao, Lanshun Nie, Dechen Zhan, Yunxin Liu, Ming Wu, and Lintao Zhang. «Efficient and effective sparse LSTM on fpga with bank-balanced sparsity». In: *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 2019, pp. 63–72 (cit. on pp. 2, 23, 25).

[16] Robert Hecht-Nielsen. «Theory of the backpropagation neural network». In: *Neural networks for perception*. Elsevier, 1992, pp. 65–93 (cit. on p. 4).

[17] Yann LeCun, Bernhard Boser, John S Denker, Donnie Henderson, Richard E Howard, Wayne Hubbard, and Lawrence D Jackel. «Backpropagation applied to handwritten zip code recognition». In: *Neural computation* 1.4 (1989), pp. 541–551 (cit. on p. 4).

[18] Michael I Jordan. «Serial order: A parallel distributed processing approach». In: *Advances in psychology*. Vol. 121. Elsevier, 1997, pp. 471–495 (cit. on p. 5).

[19] Jeffrey L Elman. «Finding structure in time». In: *Cognitive science* 14.2 (1990), pp. 179–211 (cit. on p. 5).

[20] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. «Learning representations by back-propagating errors». In: *nature* 323.6088 (1986), pp. 533–536 (cit. on p. 5).

[21] Felix A Gers and Jürgen Schmidhuber. «Recurrent nets that time and count». In: *Proceedings of the IEEE-INNS-ENNS International Joint Conference on Neural Networks. IJCNN 2000. Neural Computing: New Challenges and Perspectives for the New Millennium*. Vol. 3. IEEE. 2000, pp. 189–194 (cit. on p. 5).

[22] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016 (cit. on pp. 6, 7).

[23] Jun Han and Claudio Moraga. «The influence of the sigmoid function parameters on the speed of backpropagation learning». In: *International Workshop on Artificial Neural Networks*. Springer. 1995, pp. 195–201 (cit. on pp. 8, 10).

[24] Jürgen Schmidhuber. «Learning complex, extended sequences using the principle of history compression». In: *Neural Computation* 4.2 (1992), pp. 234–242 (cit. on p. 10).

[25] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. «Learning phrase representations using RNN encoder-decoder for statistical machine translation». In: *arXiv preprint arXiv:1406.1078* (2014) (cit. on p. 10).

[26] Shuai Li, Wanqing Li, Chris Cook, Ce Zhu, and Yanbo Gao. «Independently recurrent neural network (indrnn): Building a longer and deeper rnn». In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2018, pp. 5457–5466 (cit. on p. 10).

[27] Martin Sundermeyer, Ralf Schlüter, and Hermann Ney. «LSTM neural networks for language modeling». In: *Thirteenth annual conference of the international speech communication association*. 2012 (cit. on p. 11).

[28] Song Han, Huizi Mao, and William J Dally. «Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding». In: *arXiv preprint arXiv:1510.00149* (2015) (cit. on p. 14).

[29] Claudio Passerone. *Analog and Digital Electronics for Embedded Systems*. Vol. 1. Edizioni C.L.U.T, 2015 (cit. on p. 15).

[30] Tom Feist. «Vivado design suite». In: *White Paper* 5 (2012), p. 30 (cit. on p. 17).

[31] Grant Martin and Gary Smith. «High-level synthesis: Past, present, and future». In: *IEEE Design & Test of Computers* 26.4 (2009), pp. 18–25 (cit. on p. 17).

[32] Vivado-HLS Xilinx. *Vivado design suite user guide-high-level synthesis*. 2014 (cit. on pp. 17–19).

[33] Huan Li and Wenhua Ye. «Efficient implementation of FPGA based on Vivado high level synthesis». In: *2016 2nd IEEE International Conference on Computer and Communications (ICCC)*. IEEE. 2016, pp. 2810–2813 (cit. on p. 18).

[34] Chao Wang, Lei Gong, Qi Yu, Xi Li, Yuan Xie, and Xuehai Zhou. «DLAU: A scalable deep learning accelerator unit on FPGA». In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 36.3 (2016), pp. 513–517 (cit. on p. 22).

[35] Sicheng Li, Chunpeng Wu, Hai Li, Boxun Li, Yu Wang, and Qinru Qiu. «Fpga acceleration of recurrent neural network based language model». In: *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*. IEEE. 2015, pp. 111–118 (cit. on p. 22).

[36] Daniele Jahier Pagliari, Mario R Casu, and Luca P Carloni. «Accelerators for breast cancer detection». In: *ACM Transactions on Embedded Computing Systems (TECS)* 16.3 (2017), pp. 1–25 (cit. on p. 22).

[37] Andre Xian Ming Chang and Eugenio Culurciello. «Hardware accelerators for recurrent neural networks on FPGA». In: *2017 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE. 2017, pp. 1–4 (cit. on p. 22).

[38] Yijin Guan, Zhihang Yuan, Guangyu Sun, and Jason Cong. «FPGA-based accelerator for long short-term memory recurrent neural networks». In: *2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE. 2017, pp. 629–634 (cit. on p. 22).

[39] Joao Canas Ferreira and Jose Fonseca. «An FPGA implementation of a long short-term memory neural network». In: *2016 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. IEEE. 2016, pp. 1–8 (cit. on p. 22).

[40] Aydin Buluç, Jeremy T Fineman, Matteo Frigo, John R Gilbert, and Charles E Leiserson. «Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks». In: *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*. 2009, pp. 233–244 (cit. on p. 23).

[41] Daniele Jahier Pagliari, Roberta Chiaro, Yukai Chen, Enrico Macii, and Massimo Poncino. «Optimal Input-Dependent Edge-Cloud Partitioning for RNN Inference». In: *2019 26th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*. IEEE. 2019, pp. 442–445 (cit. on p. 23).

[42] Daniele Jahier Pagliari, Francesco Panini, Enrico Macii, and Massimo Poncino. «Dynamic Beam Width Tuning for Energy-Efficient Recurrent Neural Networks». In: *Proceedings of the 2019 on Great Lakes Symposium on VLSI*. 2019, pp. 69–74 (cit. on p. 23).

[43] Daniele Jahier Pagliari, Francesco Daghero, and Massimo Poncino. «Sequence-To-Sequence Neural Networks Inference on Embedded Processors Using Dynamic Beam Search». In: *Electronics* 9.2 (2020), p. 337 (cit. on p. 23).

[44] Marco Storace and Tomaso Poggi. «Digital architectures realizing piecewise-linear multivariate functions: two FPGA implementations». In: *International Journal of Circuit Theory and Applications* 39.1 (2011), pp. 1–15 (cit. on p. 31).

[45] Daniele Bagni, A Di Fresco, J Noguera, and FM Vallina. «A zynq accelerator for floating point matrix multiplication designed with vivado hls». In: *Application note* (2016) (cit. on pp. 39, 41).