FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Customized Hardware for Long-Short Term Memory Networks in Embedded Systems

Pedro Manuel Afonso Costa

Mestrado Integrado em Engenharia Eletrotécnica e de Computadores

Supervisor: João Canas Ferreira

February 20, 2020

Resumo

Redes neurais do tipo Long Short-Term Memory (LSTM) têm sido aplamente usadas nos últimos anos, cuja implementação se tem dado em diversos campos, tais como reconhecimento de fala ou descodificação de gestos em teclado. Adicionalmente, tem-se verificado uma necessidade crescente de implementar LSTMs em ambientes de baixa potência, nos quais ASICs e FPGAs revelam ser as melhores soluções.

Paralelamente, tem-se assistido a notáveis desenvolvimentos no que toca às plataformas de desenvolvimento em Síntese de Alto Nível (HLS), conduzindo a melhores resultados no que toca às descrições de Register Transfer Level (RTL). Por este motivo, as plataformas de HLS são cada vez mais consideradas uma alternativa viável a descrições em Linguagens de Descrição de Hardware (HDL), visto ter-se tornado possível a obtenção de circuitos de qualidade, ao mesmo tempo que são aproveitadas as vantagens resultantes da utilização de linguagens de alto nível, nomeadamente o menor esforço de implementação.

Considerando os aspectos acima, este trabalho propõe o desenvolvimento de uma rede LSTM para uma plataforma FPGA, com recurso a ferramentas HLS. Devido à dimensão elevada das redes LSTM comparativamente com a memória interna de uma FPGA, é proposto o armazenamento dos pares de entrada e dos pesos da rede em memória externa. Para evitar as elevadas latências associadas à utilização exclusiva de memória externa, é proposta uma técnica do tipo "block-batching", a qual permite efectuar operações sobre um conjunto de pares de entrada e um bloco de matrizes de pesos, os quais são transferidos de memória externa para memória interna através de "double-buffering". Por forma a obter um maior nível de paralelismo, bem como a reutilização dos elementos instanciados na FPGA, a implementação dos blocos com maiores necessidades computacionais recorre à utilização de *pipelines*. É também possível variar a topologia da rede instanciada através de um conjunto de parâmetros. A dimensão em *bits* de cada *word* pode ser ajustada de acordo com as necessidades do sistema.

Aquando da utilização do *dataset* MNIST, a arquitectura proposta utiliza uma menor dimensão de *word* para as suas variáveis, a qual resulta numa redução dos requisitos de memória interna até 1.14x comparativamente a trabalhos anteriores, incorrendo num aumento do erro de até 2.2%. Após implementação do acelerador proposto numa placa física, melhorias de 1.99x e 10.53x na velocidade de execução são registadas comparativamente a processadores *desktop* e *embedded*, respectivamente.

Palavras-chave: FPGA, HLS, LSTM, Redes Neuronais

Abstract

Long Short-Term Memory (LSTM) neural networks have been widely used in recent years, being deployed in a range of scenarios such as speech recognition or keyboard gesture decoding. In addition to this, there is an increasing necessity of implementing LSTMs in environments with low power budgets, for which ASICs and FPGAs are the most sensible solutions.

Parallel to this, High-Level Synthesis (HLS) development platforms have markedly improved in recent years, resulting in better Register Transfer Level (RTL) descriptions. HLS is then increasingly seen as a feasible alternative to Hardware Description Language (HDL) descriptions, as it is now possible to obtain quality circuits with the advantages that derive from using higher-level programming languages, namely its lower implementation effort.

Considering this, the present work proposes the development of an LSTM network for an FPGA platform using HLS tools. Because of the large size of LSTM networks in comparison with the on-chip memory available in an FPGA, off-chip memory storage of the input pairs and the weight matrices is proposed. To avoid dealing with large latencies derived from exclusively using off-chip memory, a block-batching technique is presented, which performs computations over a batch of input pairs, and a block of weight matrices, which are buffered from off-chip into the on-chip memory using double-buffers. For obtaining a higher level of parallelism and the reuse of the elements instantiated on fabric, the implementation of the most computationally-intensive blocks is performed using pipelines. Furthermore, it is possible to vary the topology of the network by using a number of parameters for that purpose. The word bit-widths can also be adjusted according to the needs of the system.

When using the MNIST dataset, the proposed architecture uses a smaller word bit-width, resulting in a reduction of up to 1.14x of on-chip memory requirements in comparison with previous works, while incurring a maximum error of up to 2.2%. After implementing the proposed accelerator on a board model, speed-ups of 1.99x and 10.53x are registered, respectively, in comparison with a desktop and an embedded CPU.

Keywords: FPGA, HLS, LSTM, Neural Networks

Agradecimentos

Gostaria de agradecer a todos os professores com quem me cruzei durante o meu percurso académico, sem os quais não seria possível a minha formação, nomeadamente os desta Faculdade de Engenharia. Agradeço também ao Prof. João Canas Ferreira pela proposta que realizou, e por ter colocado, através da sua orientação, a sua experiência ao meu serviço.

Agradeço também o apoio dos meus amigos durante as várias fases da minha vida, e das inúmeras pessoas com que me cruzei e que me ajudaram durante o curso. Agradeço também aos meus colegas de quarto durante o meu intercâmbio em Delft, Holanda, pelo que me ensinaram sobre tolerância e sobre manter a vida em perspectiva.

Os meus agradecimentos não estariam completos sem mencionar a minha família mais próxima. Aos meus pais, e à minha avó, agradeço-lhes o seu amor incondicional e o apoio incansável, com o qual pude contar quando mais necessitei. Ao meu irmão, agradeço-lhe a sua boa-vontade em ajudar-me durante o meu percurso, e a sua orientação inestimável.

Pedro Manuel Afonso Costa

"Stay hungry, stay foolish."

Steve Jobs

Contents

List of Figures

List of Tables

Abbreviations

ANN	Artificial Neural Network
ASIC	Application-Specific Integrated Circuit
AXI	Advanced eXtensible Interface
BRAM	Block RAM
CORDIC	COordinate Rotation DIgital Computer
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
DSP	Digital Signal Processor
FF	Flip-Flop
FPGA	Field-Programmable Gate Array
GOP/S	Giga Operations per Second
GPRNG	Gaussian PRNG
GPU	Graphics Processing Unit
HDL	Hardware Description Language
HLS	High Level Synthesis
IC	Integrated Circuit
II	Initiation Interval
I/O	Input/Output
IP	Intellectual Property
JTAG	Joint Test Action Group
LFSR	Linear Feedback Shift Register
LSTM	Long-Short Term Memory
LUT	Look-Up Table
ML	Machine Learning
PLAN	Piecewise Linear Approximation
PRNG	Pseudo-Random Number Generator
RAM	Random Access Memory
RNN	Recurrent Neural Network
RTL	Register Transfer Level
SIMD	Single Instruction, Multiple Data
SoC	System-on-a-Chip
VHDL	Very High Speed Integrated Circuit HDL

Chapter 1

Introduction

In this work, a hardware implementation of an LSTM network is presented. This accelerator is implemented on an FPGA and designed for manipulating large networks. For the purpose, a block-batching technique and off-chip memory storage are used. This accelerator also allows the manipulation of the internal bit-widths which, for the MNIST dataset, results in a reduction in memory requirements of up to 1.14x in comparison with previous works. After implementation on a board model, speed-ups of 1.99x and 10.53x are registered upon comparison with a desktop and an embedded CPU, respectively.

1.1 Background

Long Short-Term Networks are a popular type of Recurrent Neural Networks. Introduced by Hochreiter and Schmidhüber in 1997, LSTMs are a state-of-the-art algorithm for capturing sequences and time-series, using for that purpose a memory controller that retains long-term dependencies. As a result, LSTMs are now used in a wide range of applications on different types of hardware, from speech recognition algorithms to keyboard gesture decoders.

LSTM networks have been implemented in various computation environments. Notably, several CPU and GPU implementations are available, which are suitable for computing huge amounts of data with good performance. However, other approaches are needed when facing power constraints. In this regard, solutions using ASICs or FPGAs can be helpful for implementing LSTMs due to their good power-performance balance. Specifically, FPGAs can be reprogrammed, thus opening up the possibility of altering the system on-the-fly.

Parallel to this, HLS software has registered notable improvements in recent years, to the extent that it is now a feasible platform to produce fast, efficient circuits. With it, the use of C/C++, along many of its useful features, is supported for describing hardware. One example is the ability of using templates, that enable rapid generation of different functions on hardware by altering some parameters. Additionally, using the included pragmas, the programmer can perform

fine-tuning over some characteristics of the hardware, which would not be possible using high-level programming languages alone.

1.2 Motivation

Presently, a number of LSTM implementations on FPGA exist. Usually, the proposals focus on inference, and perform either data compression techniques or architectural improvements. Compression techniques have the advantage of reducing the size of the network (which translates to lower memory and computational requirements), and usually consist of some sort of pruning technique performed to the weight matrices of the LSTM, while in other cases quantisation techniques are mentioned. Architectural improvements may consist of buffering or batching techniques, and in some cases of alternative algorithms to compute the outputs of the LSTM.

However, most implementations do not enable rapid, effortless synthesis of different LSTM networks depending on the needs of the system. To enable this, the architecture needs to be able to accept a number of inputs that can be used to vary the topology of the network. Furthermore, and to the best of our knowledge, the only existing parametrisable implementation makes exclusive use of on-chip memory for storing all variables, which severely limits the dimension of the networks that can be processed.

1.3 Objective

In this work, the implementation of an LSTM network on an FPGA is proposed. For this purpose, a parametrisable accelerator is presented, which is implemented in Vivado HLS. The proposed accelerator uses a block-batching technique that buffers values from off-chip memory, and enables both effortless network topology changes and tuning of the word bit-widths of the variables. The proposed accelerator is implemented and tested on a board model.

1.4 Overview

The document is organised as follows. Chapter 2 introduces the concept of Machine Learning and the theoretical considerations behind ANNs, with special focus on LSTMs, and the topic of High Level Synthesis. Chapter 3 provides an overview of previous LSTM implementations in different research fields, and presents the state of the art in regard to implementations on FPGA. Chapter 4 presents the architecture of the accelerator proposed in this work, and describes its constituent blocks. Chapter 5 presents the obtained results.

Chapter 2

Problem Characterisation

This section introduces and characterises the main topics to be covered in this work. Section 2.1 provides a general overview of Machine Learning and its concepts. Section 2.2 briefly describes Artificial Neural Networks, from their basic building blocks to the necessary training procedures. Section 2.3 delves into Recurrent Neural Networks, a specific type of ANN, and introduces the Long Short-Term Networks that are used in this work. Section 2.4 finishes by providing an overview of High Level Synthesis hardware description, which will be used to specify the RTL implementation to be synthesised on FPGA.

2.1 Machine Learning: Overview

Machine Learning (ML) is a research field of Computer Science that uses statistics and artificial intelligence to enable computer systems to learn and improve from experience without being explicitly programmed [1]. This allows ML to extract latent features from the data, and enabling its classification into a particular class by using an adaptive model that adjusts its parameters according to the input data received.

Adopting ML solutions is useful in situations where hard-coded, rule-based algorithms may fail or be too cumbersome, such as in situations where small changes in a task require significant modifications to the code, or where defining rules would require a deep understanding of the subject. As a result, several ML algorithms have been developed, and most can be split into the categories below.

Supervised Learning

Supervised learning algorithms are able to automate decision-making processes by generalising from known samples. To achieve this, the algorithm is trained with pairs of inputs, or **targets**, and desired outputs, or **labels**, which constitute the **training set** and are used by the algorithm so that it finds a way to reproduce the input and output pairs. After the aforementioned training,

the algorithm is able to produce an output, given a new input, without human intervention. These algorithms are usually the easiest to understand and evaluate.

A typical supervised learning task is **classification**, which provides an output corresponding to a discrete number of solutions, of which spam e-mail classification (i.e. spam or not spam) is an example [2]. Another typical task is **regression**, where a target value in a continuous range is inferred by using predictors, of which house pricing prediction (i.e. the value of a house according to a set of features) is an example.

Unsupervised Learning

Unsupervised learning algorithms are characterised by receiving unlabelled data (i.e. only the input pairs are given, thus the outputs are unknown). Such algorithms are usually harder to understand and evaluate. An example of this is customer segmentation according to similar product preferences [1].

Other categories

Besides supervised and unsupervised learning, there are a number of other noteworthy ML algorithm categories.

Semi-supervised learning algorithms use a mixture of a small amount of labelled data and a large amount of unlabelled data in order to improve learning accuracy.

Reinforcement learning uses a learning system to observe the environment, select and perform tasks, so that it gets positive or negative rewards that make the algorithm learn by itself the best strategy to get the most rewards over time. This strategy, or policy, defines what action to choose in a given situation [2].

Recommender systems seek to predict the preferences of users, and can be built using a number of approaches, such as content-based recommendations or collaborative filtering. Such systems are often used in commercial applications, such as Amazon's product recommendations based on previous purchases, or Spotify's music suggestions based on prior listening.

2.2 Artificial Neural Networks

Artificial Neural Networks are mathematical structures that loosely resemble the neural networks found in the brain. They make up a collection of simple computational units interlinked by a system with a variable number of connections [3].

2.2.1 Perceptron

Following this line of thought, the basic element of an ANN is represented by a logistic unit, the **perceptron** [2]. It is fed with **inputs** $[x_1, x_2, ..., x_n]$ and produces an output, or **activation value**,

h_W accordingly. This is achieved by using $[w_1, w_2, ..., w_n]$ as multiplication **weights**, and a **bias** value b, as in Equation 2.1.

$$h_W = b + \sum_{i=1}^{n} w_i \cdot x_i \tag{2.1}$$

The computations above, comprising the weighted sum of inputs, correspond to the **inference**, which is illustrated in Figure 2.1, with the circle representing a computational unit. Before it can occur, however, the perceptron needs to adjust the appropriate weight and bias values that will lead to a correct solution. Such procedure is called **training**.

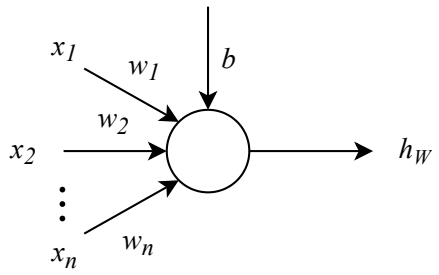


Figure 2.1: A perceptron

Besides this, the computational unit may apply an **activation function** to the right-hand side of Equation 2.1 before producing h_W. This serves as a decision threshold.

2.2.2 Activation Functions

It is desirable for an activation function, in the context of LSTMs, to produce an output that is both bounded and limited for all cases. This means that the function must have a predictable behaviour, and will not output values with large magnitudes independently of the input. Furthermore, it is desirable that the chosen function is fully differentiable. Some possibilities for activation functions follow below.

The **Heaviside step** function performs as a simple, binary-valued function with a threshold, and a discontinuity at $x = 0$ (Equation 2.2).

$$f(x) = \mathscr{H}(x) \begin{cases} 0 & x < 0 \\ 1 & x \geq 0 \end{cases} \tag{2.2}$$

The **sigmoid** function, Figure 2.2a, allows for a more complex behaviour, as it is a real-valued function, with its output moving slowly in the interval $[0, 1]$ (Equation 2.3).

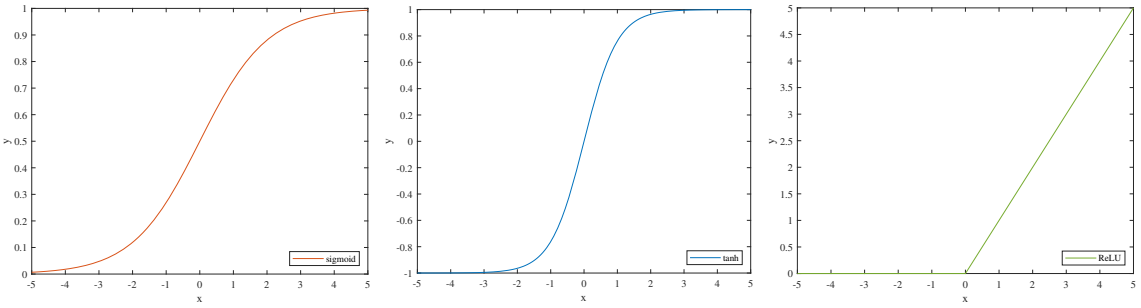
$$f(x) = \sigma(x) = \frac{1}{1 + e^{-x}} \tag{2.3}$$

The **hyperbolic tangent** function, Figure 2.2b, is very similar to the sigmoid, with its output moving slowly in the interval $[-1,1]$ (Equation 2.4).

$$f(x) = \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{2.4}$$

The **rectified linear unit** function, Figure 2.2c, introduces a non-linearity at $x = 0$, which can be used for decision making (Equation 2.5).

$$f(x) = \text{ReLU}(x) = \begin{cases} 0 & x < 0 \\ x & x \geq 0 \end{cases} \tag{2.5}$$



(a) sigmoid function (b) hyperbolic tangent function (c) rectified linear unit function

Figure 2.2: Examples of activation functions

Concerning the criteria introduced for choosing an activation function, it is concluded that both the sigmoid and the hyperbolic tangent are good candidates for activation functions. This is supported by current work in the literature on LSTMs, which often uses both the sigmoid and the hyperbolic tangent functions. As for the Heaviside function, it is not differentiable, thus it was not chosen. The rectified linear unit function, despite being widely used in other ANNs, has little used in LSTMs, as its output is neither bounded nor differentiable [4].

2.2.3 Forming ANNs

As mentioned earlier, the perceptron can be used as a basic building block to form increasingly complex models, resulting in an ANN, as depicted in Figure 2.3. The main advantage of doing this is the ability of mixing results from different inputs, thus obtaining a model that performs complex computations while using simple linear operations.

Specifically, this is performed by chaining perceptrons to one another, usually in a directed acyclic graph. The perceptrons are organised in groups of nodes occupying identical hierarchical positions called **layers**, in which the output from each neuron is used as an input to all the neurons in the following layer.

Neurons in ANN implementations are usually organised into the following layers:

- **Input layer:** accepts all the inputs and performs the first round of computations;

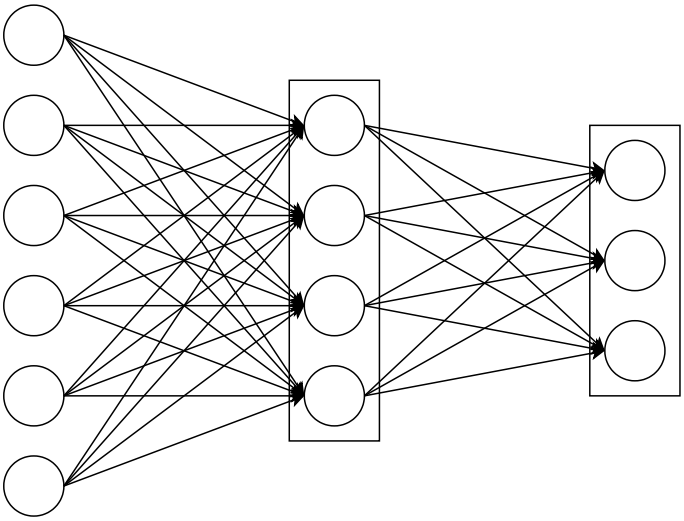


Figure 2.3: An ANN with three connected layers. Its input layer accepts 6 inputs, its hidden layer contains 4 perceptrons, and its output layer generates 3 different values.

- **Hidden layers:** perform intermediate computations; there is no restriction to the number of hidden layers, nor to the number of perceptrons in each layer;
- **Output layer:** performs final computations and outputs an intelligible value.

Each layer, similarly to a perceptron, contains its set of weights, which are used during inference mode to predict an output based in the input pairs given to it.

2.2.4 Training

As mentioned in subsection 2.2.1, the ANN needs to be trained before being able to accurately perform inference. In line with the scope of this work, the training procedure explained below focuses on a supervised learning algorithm using labelled samples.

During training, all the elements of the training set are traversed, and each of these passes is called a **training epoch**. The duration of each epoch varies depending on the input length of the training set and on the type of network.

Before any training epoch occurs, the network weights are initialised, preferably to a set of small random values. Afterwards, each training epoch is composed by a two-step process. First, **forward propagation** is performed, and then **backward propagation** occurs.

Forward propagation begins by computing all the activation values of the network. The activation values for the output layer are then compared against the existing labels using a **cost function** for measuring the accuracy of the network. A common choice is the *mean-squares error* function in Equation 2.6, which averages the error for all m training samples and for each K output class, with $(h_W(x^{(i)}))_k$ representing the predicted value for each sample.

$$J(W) = -\frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{K}\left[y_k^{(i)}\log((h_W(x^{(i)}))_k) + (1-y_k^{(i)})\log(1-(h_W(x^{(i)}))_k)\right] \tag{2.6}$$

Backward propagation then uses these values to minimise the cost function by optimising the weight parameters. At this stage, the **gradients**, which give the direction of maximum change of the cost function for each weight, are computed for each node, and determine the influence of each weight in the total error. The weights are then adjusted from outputs to inputs, and the magnitude of this adjustment tuned by changing the **learning rate**. The learning rate consists of a tuning parameter that determines the step size in each iteration, with the goal of moving towards a minimum of the cost function. The choice of this parameter is relevant since, if it is too large, the function may never converge, whereas if it is too small, convergence might take many iterations and needlessly increase computation times.

A commonly used algorithm for backward propagation is the *gradient descent*. This algorithm, which is exemplified in Algorithm 1, updates the weights iteratively after starting from a random position. Backward propagation can be performed over each sample separately, or in a batch of samples (that is, a number of samples bundled together), which can simplify computations.

2.3 Recurrent Neural Networks

2.3.1 Overview

A Recurrent Neural Network (RNN) can be seen as a directed cyclic graph, meaning that the output from one RNN cell is given as an input to another cell (that is essentially a copy of itself), as depicted in Figure 2.4, This structure, which is suitable to manipulate sequences and lists, enables RNNs to deal with information that contains a sequence (i.e. that depends of previous inputs), such as speech recognition or natural language processing.

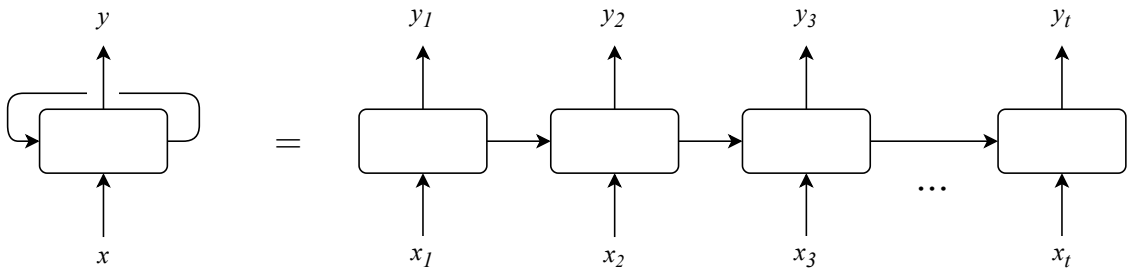


Figure 2.4: An RNN and its unrolled equivalent [5]

RNNs can be trained using a number of algorithms, *backward propagation through time* being a common solution. This is similar to backward propagation in subsection 2.2.4, but now taking into account the unrolling of the network through time.

The output data obtained at time t, y_t, is mainly influenced by data obtained immediately prior to it, as earlier values cannot exert much influence. This happens because of the difficulty, during training, for the error at y_t to backpropagate to the initial values of the sequence. This represents the **vanishing gradients** problem, which also occurs with ANNs. Specifically, in the context of

RNNs, this problem results in an increasing difficulty of an RNN to relate long dependencies between present and past information.

Algorithm 1 Backward propagation using gradient descent for the network in Figure 2.3

Training set: $\{x^{(i)}, y^{(i)}\}$
Activation value: $a_j^{(l)}$
Output value: $z_j^{(l)}$
Weight matrix value: $W_j^{(l)}$
Gradient: $\delta_i^{(l)}$
Gradient accumulator: $\Delta_{ij}^{(l)}$
Learning rate: λ

for all pair i, node j, layer l **do**
  $\Delta_{ij}^{(l)} \leftarrow 0$
end for

for all i of m in training set **do**
  function FORWARD PROPAGATION
    for all j nodes in layer **do** ▷ Computed from input to output
      $a_j^{(1)} \leftarrow x_j^{(i)}$
      $z_j^{(2)} \leftarrow W_j^{(1)} a_j^{(1)}$
      $a_j^{(2)} \leftarrow h_W(z_j^{(2)})$ (add $a_0^{(2)}$)
      $z_j^{(3)} \leftarrow W_j^{(2)} a_j^{(2)}$
      $a_j^{(3)} \leftarrow h_W(z_j^{(3)})$
    end for
  end function

  function BACKWARD PROPAGATION
    for all j nodes in layer **do** ▷ Computed from output to input
      $\delta_i^{(3)} \leftarrow a_j^{(3)} - y_j^{(i)}$
      $\Delta_{ij}^{(2)} \leftarrow \Delta_{ij}^{(2)} + \delta_i^{(3)} (a_j^{(2)})^T$
      $\delta_i^{(2)} \leftarrow (W_j^{(2)})^T \delta_j^{(3)} \circ (a_j^{(2)} \circ (1 - a_j^{(2)}))$
      $\Delta_{ij}^{(1)} \leftarrow \Delta_{ij}^{(1)} + \delta_i^{(2)} (a_j^{(1)})^T$
    end for
  end function
end for

for all l layers **do**
  $D_{ij}^{(l)} \leftarrow \frac{1}{m}(\Delta_{ij}^{(l)} + \lambda W_{ij}^{(l)})$, $j \neq 0$
  $D_{ij}^{(l)} \leftarrow \frac{1}{m}(\Delta_{ij}^{(l)})$, $j = 0$
end for

2.3.2 Long Short-Term Memory Networks

Long Short-Term Memory Networks (LSTMs) are a variation of RNNs that use a memory controller to be able to retain long-term dependencies [6].

The architecture of an LSTM cell is depicted in Figure 2.5. The core idea behind these networks is the cell state C_t, which serves as a memory element [5]. This state is controlled by four network gates that have the ability to let information through:

- **Forget gate, f_t:** decides whether to keep or discard information in the cell state C_{t-1}, thereby controlling the cell state;
- **Input gate, i_t:** decides which values to be updated;
- **Update gate, z_t:** decides the candidate values (i.e. the flow of information) to be added to the cell state C_t;
- **Output gate, o_t:** decides which values (i.e. the flow of information) to output to h_t.

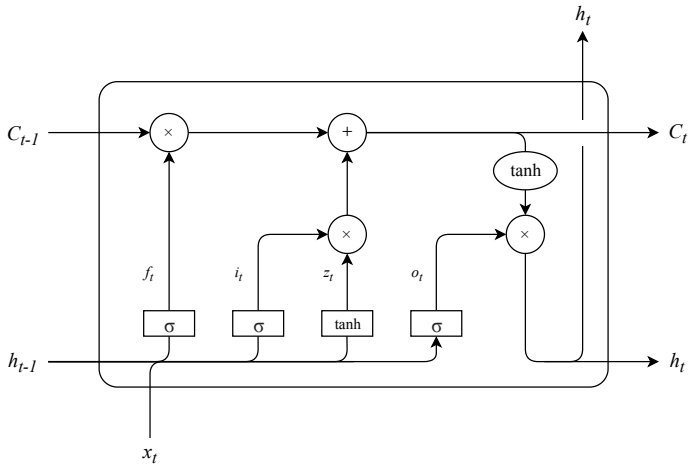


Figure 2.5: An LSTM cell [5]. Rectangles show neural network layers and their corresponding activation functions, and circles denote a pointwise ($\odot$) operation. Arrays are represented by lines, with array concatenation and copying shown by line merging and forking, respectively.

The hidden state h_t serves as the output of the cell. It can be passed through a *softmax* filter, which returns the output probabilities. By retrieving the index of the maximum probability value out of the *softmax* layer (or by simply returning the maximum value of h_t), it is possible to determine the predicted value.

The update gate z_t also shields the LSTM from the vanishing gradients problem. Because the cell state C_t can acquire values very close to 0 or 1, the existing information in it is able to persist in the system for many iterations.

There is a number of possibilities to formulate the equations for inference with an LSTM. In Equation 2.7, a solution that maximises parallel computations at time t is shown.

$$\begin{aligned} f_t &= \sigma([\mathbf{W}_{if}x_t + \mathbf{b}_{if}] + [\mathbf{W}_{hf}h_{t-1} + \mathbf{b}_{hf}]) \\ i_t &= \sigma([\mathbf{W}_{ii}x_t + \mathbf{b}_{ii}] + [\mathbf{W}_{hi}h_{t-1} + \mathbf{b}_{hi}]) \\ z_t &= \tanh([\mathbf{W}_{iz}x_t + \mathbf{b}_{iz}] + [\mathbf{W}_{hz}h_{t-1} + \mathbf{b}_{hz}]) \\ o_t &= \sigma([\mathbf{W}_{io}x_t + \mathbf{b}_{io}] + [\mathbf{W}_{ho}h_{t-1} + \mathbf{b}_{ho}]) \\ c_t &= f_t \odot c_{t-1} + i_t \odot z_t \\ h_t &= o_t \odot \tanh(c_t) \end{aligned} \tag{2.7}$$

All weights and biases are highlighted in bold. Defining N as the input dimension, M as the output dimension, and H as the dimension of the hidden layer, its description follows:

- $\mathbf{W}_i$ matrices represent weights associated to the input pairs x_t, and contain $H \times N$ elements
- $\mathbf{W}_h$ matrices represent weights associated to the hidden state h_{t-1}, and contain $H \times H$ elements
- $\mathbf{b}_i$ vectors represent bias values associated to the input pairs x_t, and contain H elements
- $\mathbf{b}_h$ vectors represent bias values associated to the hidden state h_{t-1}, and contain H elements

The usage of the hidden state can vary depending on the requirements of the network. It is possible to randomly generate a hidden and cell state vector for every input pair prediction, which constitutes a stateless network. Another option consists of using the hidden and cell state vectors to compute the new predicted values, which constitutes a stateful network.

A number of modifications have been suggested throughout the years. Namely, the introduction of a forget gate [7] and peepholes have been suggested. Upon considering their usefulness, the former will be used in this work, whereas the latter will be discarded, as it has been proved to have a minimal impact on performance [8] while requiring greater resources.

2.4 High Level Synthesis

2.4.1 Overview

Traditionally, programming on FPGAs requires the use of a Hardware Description Language (HDL) such as Verilog or VHDL, which is then translated to Register Transfer Level (RTL) that specifies the design using parallel processes that operate on vectors of binary or simple data type signals. With the rapid increase of complexity in System-on-a-Chip (SoC) designs, the need arose for using design abstractions that were faster and more effective to implement than RTL.

This motivated the development of High Level Synthesis (HLS) tools that could use a higher-level programming language (such as C/C++) to specify a synthesisable RTL implementation [9]

for ASICs and FPGAs while hiding several implementation details. A HLS tool thus provides a programming development environment more similar to that of standard processors.

In recent years, Xilinx developed a tool for this purpose, named Vivado HLS. This tool accepts C, C++ or SystemC (which is a subset of C++), however C++ code interpretation is optimised. The code is then translated to HDL and described at RTL level. The source code can be compiled and verified using tools written for C/C++ for interpreting, analysing, and optimising the code.

When it comes to writing C/C++ code, Vivado HLS tools are similar to those of processor compilers for interpretation, analysis and optimisation of programmes, while targeting FPGA systems. Because of this, application code for Vivado HLS is similar to standard C/C++ code (with the exception of dynamic memory allocation, which is not supported because of the memory architecture of the FPGAs), so it can normally analyse operations, conditional statements, loops, and functions [10]. Variables, classes and structs can be assigned to registers, whereas arrays are mapped to BRAM structures. Besides this, code in C/C++ can be synthesised without explicitly declaring any clock signal. These characteristics abstract the programmer from several low-level details and make HLS easier to use in comparison with HDL tools.

Despite this, programming in Vivado HLS is a more challenging task than C/C++ programming for software, while providing less control over the generated hardware than traditional HDL implementations. Although a large subset of C/C++ code that is not optimised for hardware is often synthesisable and functional, it usually results in slow, cumbersome RTL code, due to the inherent differences between general-processing and parallel programming. Thus, in order to obtain efficient, synthesisable code, the programmer needs to take into account the hardware structure of the FPGA and programme accordingly. In some instances, optimisations are highly dependent on the coding style, as the HDL output can only be optimised (or at all used) by Vivado HLS if it obeys to some canonical form described by Xilinx. Besides this, there are a number of code optimisations that can only be performed by programmer-defined **pragmas**. Pragmas are directives used by Vivado HLS to perform local code optimisations. These are instantiated in the code or by a directives script. This means that, in order to obtain good results under HLS, the programmer needs to have a grasp not only of the underlying hardware, but also of the specific structures and optimisations that can be performed by the HLS software that is being used.

2.4.2 Pragmas

With the goal of directing further performance and area optimisations, Vivado included a set of optimisation directives, called pragmas [9]. The relevant options to be used or related with this work are explained below.

Array Partitioning

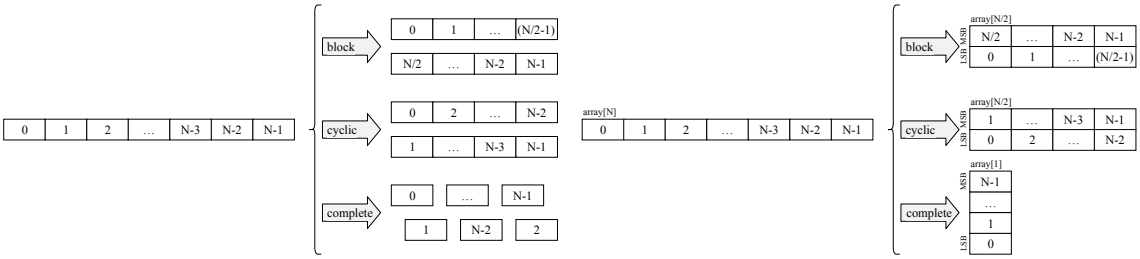
***#pragma** HLS ARRAY_PARTITION variable=<name> type <factor=N> dim=<N>*[1] and

[1] *factor* specifies in how many blocks the array is split into; *dim* specifies the dimension in which the transformation is performed, thereby allowing the programmer to optimise the access of the array for a specific dimension.

***#pragma** HLS ARRAY_RESHAPE variable=<name> type <factor=N> dim=<N>*

These directives can modify the arrangement of arrays mapped into memory. Vivado HLS defaults array implementation to BRAM, which has a maximum of two data ports. This often constitutes a bottleneck by limiting the number of read and write instructions possible per clock cycle. With these directives, Vivado provides three types of array partitioning, as depicted in Figure 2.6:

- **Block:** division into equally sized blocks of consecutive elements of the original array
- **Cyclic:** division into equally sized blocks by interleaving the elements of the original array
- **Complete:** fully splits the array into its individual elements by placing them in individual registers



(a) array partitioning (b) array reshaping

Figure 2.6: Array transformations provided by Vivado HLS [9]

ARRAY_PARTITION and *ARRAY_RESHAPE* differ solely on the method used for this partitioning. Whereas the former opts for physically splitting the arrays into multiple physical arrays in order to obtain more read/write ports, the latter allows parallel access of data via *vertical mapping* of the words, which bundles several elements of the array into a single element with a larger bit-width.

Interfaces

***#pragma** HLS INTERFACE mode port=<name> <bundle=string> register <register_mode> <depth=N> <offset>*

It provides the ability to specify how the RTL ports from the function description are created. By default, Vivado HLS defaults *mode* to `ap_ctrl_hs`, which is used for block-level I/O and implements a handshake protocol (i.e. it indicates when to start design operation, and when the design is idle, done, and ready for new input data). On the top-level, it also supports AXI interface modes [11]:

- **AXI4-Stream (`axis`):** defines a single channel for transmission of streaming data that can be used to burst an unlimited amount of data, and is ideal for transferring streams of data (e.g. audio, video file);

- **AXI4-Lite (`s_axilite`):** similar to AXI4-Master (but without burst support), it can be used to transfer small amounts of data (e.g. a parameter in a variable);
- **AXI4-Master (`m_axi`):** used for transferring small amounts of data, and preferred for transferring bursts of data with high transfer speeds (e.g. a parcel of a wide array of data). The usage of `m_axi` requires specifying the *depth* of the FIFO used in simulation, which corresponds to the amount of data accesses to the array, otherwise RTL Co-Simulation will not work properly.

Using the AXI4 protocol can simplify the connection of an IP block with other elements in the architecture due to its wide support. Moreover, Vivado HLS automatically generates device drivers for managing IP blocks with AXI4-Lite ports. By using the *offset* option, this can be used to specify the starting pointer of an array, which can then be used by the accelerator to access DDR memory according to its needs.

Data and Control Flow

***#pragma** HLS LOOP_FLATTEN <off>*

It enables nested loops to be collapsed into a single loop with improved latency, thereby saving clock cycles (entering and exiting a loop in RTL requires 1 clock cycle). It can only be used with loops where only the innermost loop has body content, and where all loop bounds are constants (however, the outermost loop can be a variable).

***#pragma** HLS UNROLL <factor=N> <region> <skip_exit_check>*

It enables loop unrolling, either completely or partially by a factor of N. This enables all loops to execute in parallel, however this is only possible if no data dependencies exist between different iterations.

***#pragma** HLS PIPELINE <II=N> <enable_flush> <rewind>*

It enables function and loop pipelining, however, only loop pipelining will be described due to the scope of this work. It tries to implement a design with an Initiation Interval (II), which consists of the number of clock cycles between the start times of consecutive loop iterations, specified by the programmer. It defaults to 1, and if the value cannot be achieved, Vivado HLS tries to implement a design with the minimum II possible. Loop pipelining allows the operations in the loop to overlap, thus leading to significant reductions in the number of clock cycles during sequential operation. The scenario depicted in Figure 2.7 illustrates the results achieved with pipelining: whereas the sequential version has an II of 3, and requires 8 cycles until the last output is written, the pipelined version has an II of 1, and requires only 4 cycles to write the last output. One caveat of this directive is that, despite only pipelining the specified region, it forces loop unrolling over all nested loops below the pipeline. Furthermore, using pipelines in Vivado HLS requires the use of constants for all loop bounds.

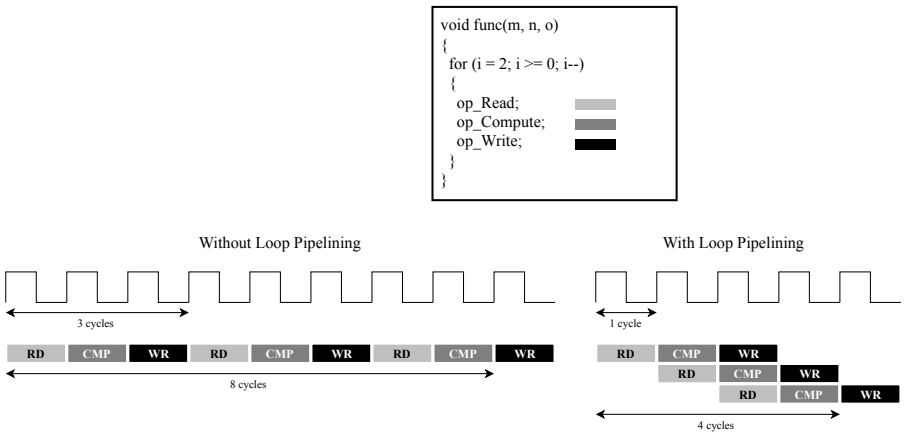


Figure 2.7: Loop pipelining with Vivado HLS [9]

***#pragma** HLS INLINE <option>*

It removes a function as a separate entity in the hierarchy and inserts (i.e. inlines) it in whatever code block it is called, which in some cases enables operations within the function to be shared and optimised more effectively (e.g. by reducing function call overhead). This is often performed automatically for small functions by Vivado HLS. An *off* option is provided, which is convenient for large functions. With it, block interfaces are clearly defined, which allows for an easier tracking of data dependencies, thereby enabling greater block-level parallelism.

2.4.3 Data Types

Vivado HLS provides a number of libraries to help during design implementation. Namely, it provides a set of libraries that enable the creation of customised datatypes.

The most important of these is `ap_fixed.h`. It enables the definition of fixed-point data types according to the needs of the design, using the form *ap_[u]fixed<W,I,Q,O,N>*. The explanation for each parameter, as per [9], follows below.

- **W:** number of bits for the word length
- **I:** number of bits for the integer part
- **Q:** quantisation mode for when greater precision is generated than can be defined by the decimal component of the variable
- **O:** overflow mode for when a value that exceeds the possible representation is achieved
- **N:** number of saturation bits in overflow wrap modes

A library for integers, `ap_int.h`, is also defined. It allows the definition of signed and unsigned integers with variable bit length, using the form *ap_[u]int<W>*, with **W** defining the number of bits of the word length.

2.5 Summary

In this chapter, a number of key concepts concerning the present work have been discussed. The brief introduction to Machine Learning in Section 2.1 allowed to understand the motivation behind its widespread usage, not least its ability to learn and solve problems without explicit programming. Afterwards, the explanation on ANNs in Section 2.2, and afterwards of RNNs and LSTMs in Section 2.3, enabled the comprehension of their utility in different scenarios (namely, the adequacy of ANNs for a number of problems, and the capability of RNNs, and concretely LSTMs, to overcome the issues of ANNs when dealing with time dependencies). In Section 2.4, the concept of High Level Synthesis, which has the potential of significantly simplifying hardware synthesis by using a high-level language for creating RTL descriptions, was introduced.

Chapter 3

State of the Art

This section provides an overview of the state of the art for LSTM networks. Section 3.1 provides an overview of the potential of LSTMs, and shows some examples of work performed with this type of networks. Section 3.2 describes some notable, state-of-the-art LSTM implementations on FPGA. Section 3.3 provides a summary on the topic.

3.1 LSTM Applications: Overview

LSTM networks are currently, due to their superior performance, a state-of-the-art algorithm for a wide range of applications, namely for prediction and time-series classification of data. Therefore, significant work has been performed with this type of RNN.

To illustrate, some examples in the academia include a prize-winner handwriting algorithm specialised in unsegmented cursive writing [12], a speech recognition algorithm focused on keyword spotting [13], and a music composition algorithm using a text-based network [14]. The application of LSTM networks is also common in the industry, examples of this being a probability forecasting algorithm produced by Amazon [15], and a keyboard gesture decoding algorithm developed by Google [16].

The wide adoption of LSTM solutions, and their varying performance and resource requirements, led to their implementation in several hardware platforms.

A number of frameworks is available for use with CPUs and GPUs, such as Keras[1], PyTorch[2], and TensorFlow[3]. These implementations are usually optimised for high performance by using Single Instruction, Multiple Data (SIMD) and multi-threaded instructions on CPUs, and CUDA or OpenCL kernels on GPUs.

[1] https://keras.io
[2] https://pytorch.org
[3] https://tensorflow.org

Hardware-level solutions have also been implemented, with ASICs being used for specialised applications with a fixed network topology. Furthermore, a number of solutions on FPGA have been explored.

3.2 LSTM Implementations on FPGA

Over the last few years, a number of LSTM implementations on FPGA have been developed. Some proposals focus on optimising the LSTM network to be used by applying pruning and quantisation techniques. Briefly, the former technique consists of setting some weights of the LSTM to zero, so that they can be discarded during computations. Generally, the resulting pruned weight matrices are sparse, which is problematic for hardware as it requires non-optimal, random data accesses. Other proposals put a greater emphasis on optimisations performed at the architectural level, namely by increasing computation parallelism. With this in mind, some relevant FPGA implementations are described below.

Fonseca et al. [17] developed one of the first FPGA implementations of an LSTM. It stores the LSTM network and input pairs in on-chip memory for faster access, and uses a 18-bit fixed-point system for data representation, chosen to make full usage of the DSP slices on the FPGA. A network with a maximum hidden dimension of 256, for an input dimension of 2, was achieved on a Xilinx Zynq XC7Z045 SoC. The implementation was tested with an 8-bit adder.

Wang et al. [18] introduced a pruning solution to tackle the uneven memory accesses that arise from unstructured pruning. Moreover, a framework for implementing various LSTM variations on FPGA, C-LSTM, is presented.

The authors propose to compress the LSTM model by using block-circulant matrices, where each row vector is the circulant transformation of the row vectors. By dividing the original matrix into blocks, it is possible to greatly compress the matrix by reducing the number of parameters. However, this comes at the cost of accuracy, specially because there is no prior selection of the weights to be pruned, which might be important for the network to operate appropriately. Using circulant matrices enables the use of a Fast-Fourier Transform, thus reducing the complexity of the computations. It is important to note that this requires pre-processing the data, which will add up to the overall system latency. A double-buffering mechanism is used to improve the parallelism of computation in the system.

The C-LSTM framework presented, in turn, consists of two distinct parts, one concerning system training on TensorFlow, and another focused on implementation in FPGA that uses a scheduling graph and a code generator to produce synthesisable code, constituting a structured-compression technique.

The solution is tested on the TIMIT [19] dataset, where the authors claim to use weight matrices up to 14.6x smaller and 3.7x less computationally-intensive in comparison with the original

matrices, while incurring in a performance degradation in comparison with an uncompressed matrix of up to 1.23%.

Cao et al. [20] proposed a pruning solution with bank-balanced sparsity. In this paper, the method split each matrix row in multiple equally-sized subrows, applying weight pruning independently and obtaining the same number of non-zero values. With this, the authors use a more structured sparsity pattern, and obtain better results on hardware, and load balancing between BRAMs is achieved. The input pairs are stored in an array buffer, and partitioned in blocks to enable parallel access. This is possible thanks to the bank-balanced architecture used, which guarantees that every array has the same number of elements, and that at most one element from each block is accessed every clock cycle. Additionally, double-buffers are used to overlap data transfer and computation operations.

The authors claim that this bank-balanced sparsity method obtains higher model accuracy than block sparsity by preserving the unstructured distribution of non-zero weights for each bank, while at the same time achieving similar performance in comparison with unstructured block sparsity.

The bank-balanced sparsity solution, with a 16-bit fixed-point representation, is compared against a 32-bit floating-point representation baseline. For PTB [21] and TIMIT, prediction accuracy is maintained for up to 80% and 90% sparsity, respectively.

Wang et al. [22] dealt with network compression techniques, which make use of an algorithm previously developed by the authors, HOCA [23], and perform memory access pattern optimisations.

The HOCA algorithm consists of two steps: initialisation and training with clipped gating and quantisation, and top-k pruning. First, a technique called clipped gating is used, which sets the activation function to 0 if its value falls below a pre-determined threshold. For quantisation, a fixed-point quantisation scheme is introduced, which uses a rounding mechanism to the nearest decimal place achievable in fixed-point, and a logarithmic scheme that quantises each weight to the nearest power of two. The algorithm finishes by performing top-k pruning, which generates structured sparse matrices by concatenating all weight matrices, grouping them in sets, and then pruning until at most k non-zero elements remain in each group. The aforementioned procedures are performed on software.

A number of architectural improvements are also performed. Double-buffers are used to overlap data transfer and computation operations. Computation of the input pair and hidden state components for each gate is performed sequentially. During computation, weight values are fetched from a row at a time. While this promotes input pair value re-utilisation, as these values only need to be cached once, it results in little weight reuse, as this requires the batch size chosen to be small and dependent on the input dimension. The network is architected so that an arbitrary number of LSTM layers can be loaded.

The solution is tested on the TIMIT dataset, where the authors claim to use weight matrices up to 32x smaller and with a computation complexity up to 22.61x lower in comparison with the

original matrices, without incurring any performance degradation.

Rybalkin et al. [24] released an open-source library extension for HLS which implements a parallelisable architecture of LSTM layers on FPGA. With this library, the authors claim to enable parametrised performance scaling that offers different levels of parallelism. It uses on-chip memory with a variable-width fixed-point system. The architecture is based on a previous solution produced by the authors [25].

An LSTM with two layers is used. This approach results in increased computation parallelism, as a forward layer (i.e. reads input pairs from left to right) and a backward layer (i.e. reads input pairs from right to left) perform computations in parallel. The layers are concatenated at their outputs. Memory access patterns are rearranged to make use of this architecture. Afterwards, further processing is performed using a linear layer, whose output is passed to a max layer that determines the maximum (i.e. predicted) value. Its parametrisable architecture allows, on the one hand, to provide concurrent execution of multiple LSTM cells and, on the other hand, to divide the execution of a single LSTM cell over multiple cycles.

The HLS library extension enables usage of a fixed-point, variable-width data type for the weight matrices, input and output activations, and recurrent activations[4]. The bit-width of the input pairs is not described, although it is set at 5 bits in [25].

The solution is tested on a custom OCR dataset, with a reported test accuracy above 90% for any combination of bit width between 1-8 bits for each data type, and a throughput increase of up to 5.7x when comparing a 1-bit (i.e. binarised) to an 8-bit implementation.

Que et al. [26] presented an implementation in which the operations of the LSTM are reorganised to eliminate data dependencies. Additionally, a block-batching solution is used, which fetches partial square blocks for each weight matrix and for a batch of input pairs. A double-buffering mechanism is used to improve the parallelism of computation in the system. The system uses a 16-bit fixed-point system for data representation.

To optimise the operations, a technique is used that first performs partial computations over the input pair values, (with its data re-usage dependent on the chosen block size) without using the corresponding hidden states. The purpose of this is to allow the next inference to occur without stalling the system pipeline. In this solution, the weight matrices are not split into their input pair and hidden state components. The weight matrices are partially cached along the columns, and fully cached along the rows. The values output by the system are obtained directly from the LSTM (i.e. no fully connected layer is used).

The proposed block-batching technique, for computational purposes, combines the input pair and hidden state weight matrices (for each gate) into a single matrix, resulting in the sequential computation of each of these components. To do this, the input pair values and the previous hidden state are concatenated into a single input array. The blocking mechanism consists of splitting the weight matrices in column blocks, and then fully fetch them along the rows. In turn, the

[4]The LSTM variant used in this work implements peepholes.

concatenated input array is likewise partially fetched, and the computations for each component are performed. The batching mechanism consists of fetching several concatenated input arrays in a single memory access. The goal is to have a computation time equal or greater to the transfer time, so that memory caching is transparent (i.e. it does not impact system performance).

The solution is tested with output data from the average pooling layer of the Inception-v3 network, which was pre-trained in the ImageNet dataset. When compared with CPU and GPU solutions making use of TensorFlow[5], it is reported to provide speed-ups of up to 23.7x and 1.3x respectively, with a corresponding 208x and 19.2x reduction on power consumption. A comparison between the accuracy achieved in each of these platforms is not explicitly stated.

3.3 Summary

A summary of the characteristics of each solution is presented in Table 3.1.

Paper	Wang [18]	Cao [20]	Wang [22]	Rybalkin [24]	Que [26]
Device	Xilinx Virtex-7 XC7VX690T	Intel Arria 10 GX1150	Intel Arria 10 SX660	Xilinx Zynq XCZU7EV	Xilinx Zynq XC7Z045
Frequency	200 MHz	200 MHz	200 MHz	266 MHz	142 MHz
Description	HLS, C/C++	HDL, SystemVerilog	HDL	HLS	HDL
Storage	On-chip	Off-chip	Off-chip	On-chip	Off-chip
Data representation	16-bit fixed-point	16-bit fixed-point	8-bit fixed-point	1-to-8-bit fixed-point	16-bit fixed-point
Weight pruning	Block-circulant	Bank-balanced sparsity	HOCA architecture [23]	No	No
Architecture	FFT, double-buffer,* operation scheduler	Array partitioning, double-buffer†	Batching, double-buffer†	Variable-width data types	Block-batching, dobule-buffer†
Dataset	TIMIT	PTB, TIMIT	TIMIT	OCR‡	ImageNet

* Used to improve computation parallelism
† Used as a *ping-pong* buffer, i.e. for simultaneous computations and data transfer
‡ Custom dataset

Table 3.1: Summary of the described LSTM implementations on FPGA

In conclusion, when implementing LSTM networks on FPGA, memory storage of the weights and the input pairs is one of the limiting factors for the achieved performance. To overcome this, some proposals opt for pre-processing the weight values, by performing pruning or quantisation to reduce the size of the input pairs and weight matrices in memory. As for the storage itself, some papers opt to store the values on-chip, while others opt to store all values off-chip, and

[5]The authors used a Intel Xeon E5-2665 CPU and a TITAN X Pascal GPU.

then transferring them to the memory in the fabric on-the-go. In several instances, double-buffer techniques are used to overlap data transfer with computation operations to avoid large latencies.

Regarding weight pruning, it is a commonly-used technique both for FPGAs and for other platforms to achieve good predictions with less memory and fewer computations. Additionally, these techniques may be used on top of other architectural optimisations. Nevertheless, it commonly happens that pruning techniques do not take into consideration the influence of each weight in the final result, which may lead to removal of important values, and as a result to performance degradation of the network. Thus, pruning should be tailored depending on the network to be implemented. Besides this, it is necessary to consider the implications of using pruning on hardware due to its greater implementation complexity. For instance, in many cases weight pruning leads to random memory accesses, which are slower, and may require additional memory to store the positions corresponding to the valid weights after pruning. Because the scope of this work focuses on a general LSTM implementation, pruning techniques will not be considered.

With respect to architectural improvements, many solutions use double-buffer techniques to speed up network performance. Because of their usefulness, double-buffers will be used throughout this work.

Batching (i.e. forming a bundle of input pairs) is also commonly used, because it enables greater weight reuse, thus leading to fewer accesses to memory. Nevertheless, most techniques can only use batching to a limited extent, because weight matrices are usually fully fetched alongside one of its dimensions. This approach has two issues. First, it uses a significant amount of memory, which immediately limits the number of input pair values that can be batched to memory. Second, fully fetching a matrix in any of its dimensions requires keeping more temporary accumulators used for storing the results of the matrix-vector multiplications between the weight matrices and the input pairs without increasing system parallelism.

As for data representation, the fixed-point proposals mentioned in this chapter do not specify the position of decimal place, or the behaviour of the network when overflow occurs. Changing any of these parameters might have severe implications on the accuracy of the network.

Another important aspect consists of the limitations of the networks that can be used in these accelerators. For the accelerators in Table 3.1 that solely use on-chip storage, restrictions in terms of network size and input pair values will always be high because of the limited memory available inside an FPGA, even if reconfigurability is allowed. As for the off-chip variants, little is said about the scalability of the network except for Que et. al [26], who mention as further work the automatisation of the architecture to enable rapid development of new designs.

Additionally, most LSTM network implementations on FPGA only focus on inference. This happens for two reasons. First, the process of inference is composed mostly of matrix multiplications, which can be highly parallelised, and are ideal for implementing on an FPGA. Second, previous training implementations generally show poor performance. This happens because, when using fixed-point operators, the computed deltas contain small errors that accumulate during training. While implementing a floating-point design could be a solution, it would, at the present moment, require a large amount of resources in the fabric.

Chapter 4

Proposed Architecture

This section describes the architecture of the accelerator proposed in this work. Section 4.1 offers a high-level overview of the most relevant components of the system. Section 4.2 introduces with more detail the building blocks of the accelerator, namely initialisation, buffering, computation, and auxiliary blocks.

4.1 Overview

4.1.1 Description

The proposed FPGA accelerator consists of a hardware implementation of a single LSTM cell, followed by a fully connected layer for dimensionality reduction. Figure 4.1 depicts the complete architecture of the LSTM accelerator.

The accelerator was developed for usage with a network with arbitrary size. This means that using a set of input pairs with larger size than that available in on-chip memory is possible, since main memory storage is provided outside the reconfigurable fabric. To optimise data access, the accelerator performs buffering and computation operations in parallel. For this purpose, the accelerator uses a double-buffering technique that stores the input pairs and weight matrices of the current state on-chip, which enable fast access during computations, while simultaneously buffering the values from off-chip memory to be used in the next computation iteration.

The accelerator uses fixed-point arithmetic in its architecture. While still providing good results, this approach results in less computational resources and provides lower latency on FPGAs. Additionally, the bit-width of the variables can be tuned for input pairs, weight matrices, hidden and cell states, and output values. This can be further used to reduce computational power (i.e. by using a maximum of 18 bits, only a single DSP is needed), and memory resources, with the potential of enabling faster manipulation of large networks.

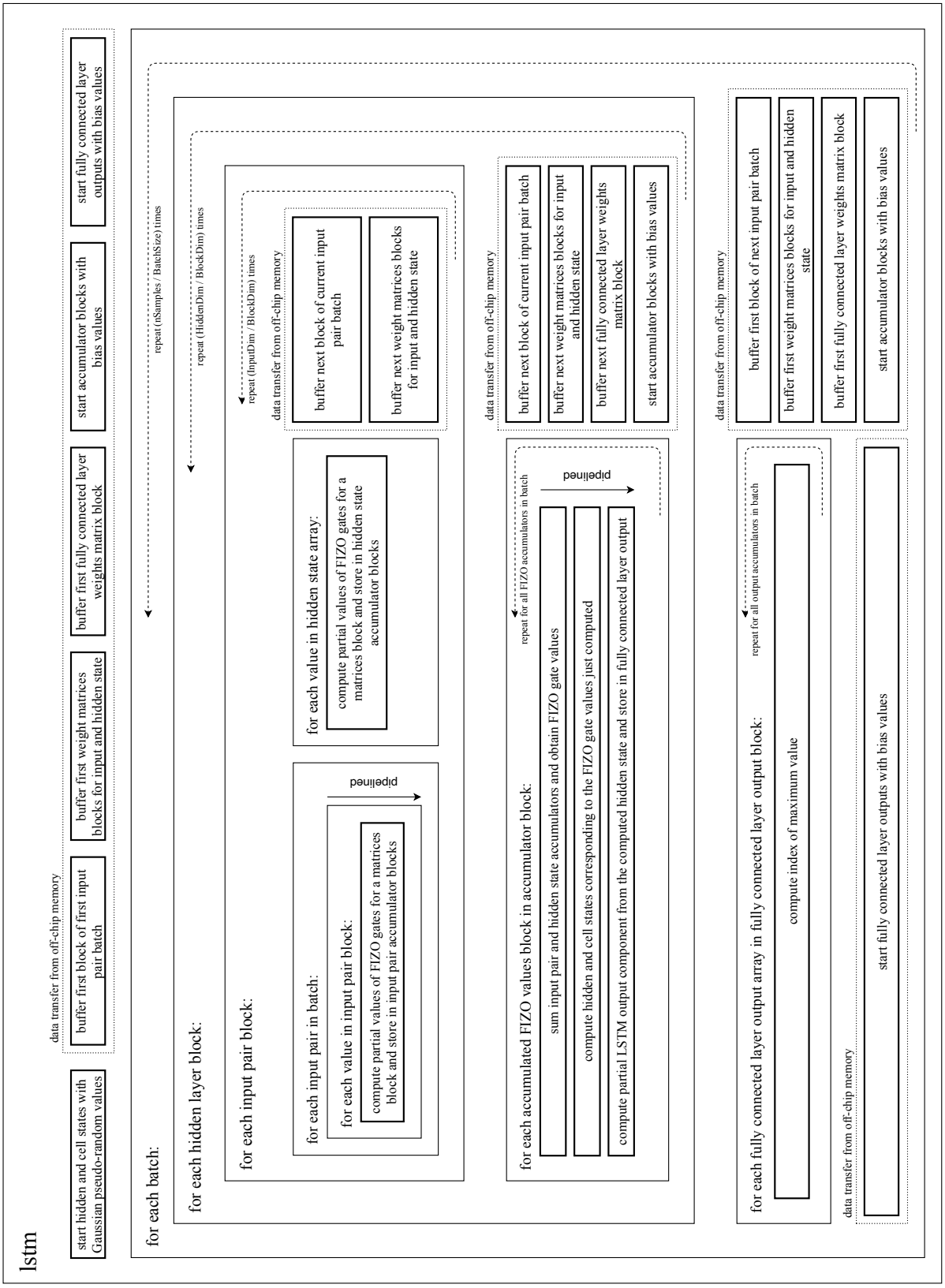


Figure 4.1: High-level architecture of the LSTM accelerator

Consistent with the reconfigurable capability of FPGAs, this accelerator makes use of C++ templates to allow effortless specification of its main parameters, with an example presented in Listing 4.1.

The parameters can be divided into the following categories:

- **Network dimensions:** refers to the input, hidden, and output dimensions of the network. This allows effortless generation of different networks;
- **Batch and block sizes:** refers, respectively, to the number of input pairs to be simultaneously processed by the accelerator, and the number of values from each input pair in a batch to be processed in one iteration. These parameters define the core functionality of the architecture, because they enable weight and input pair reuse;
- **Data types:** refers to the bit-width and integer-part width of input pairs, weight matrices, hidden and cell states, and output values.

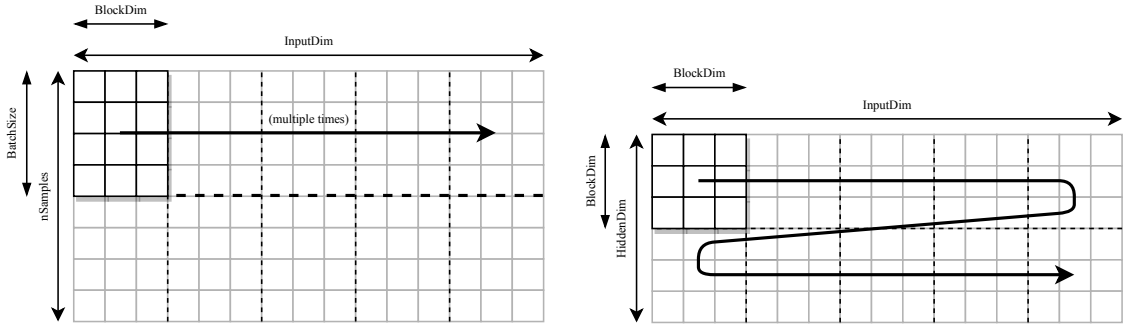
```
// Defines network dimensions
#define nSamples    500
#define InputDim    784
#define HiddenDim   128
#define OutputDim   10

// Defines batch and block sizes
#define BatchSize   500
#define BlockSize   64

// Defines length for data types
#define WidthInput  18
#define IntInput    2
#define WidthHidden 14
#define IntHidden   6
#define WidthMem    14
#define IntMem      6
#define WidthCalc   14
#define IntCalc     6
#define WidthOut    4
```

Listing 4.1: Architecture parameters with example values

4.1.2 Matrix computations

The core of the architecture consists of the strategy used to perform matrix computations. In an effort to perform data re-use when possible, a block-batching strategy with similarities to the one described in [26] was used.

This block-batching technique consists of using a partial number of input pairs, which are bundled in a batch. With this, access to off-chip memory is optimised. In turn, each buffered input pair is only partially fetched from memory, and the partial array is multiplied by the partial forget, input, update, and output weight matrix blocks. These multiplications are performed along the row of both the input pair matrix, which contains all input pair values, and the weight matrices, as depicted in Figure 4.2.



(a) input pair batch matrix traversal

(b) weight matrices traversal

Figure 4.2: Matrix traversal used in the block-batching technique

The computation results are stored in accumulators storing the temporary values for the input pair components of the forget, input, update, and output gates, which, after a full pass through the weight matrices rows, are used in the sigmoid and hyperbolic tangent functions to determine their value for each input pair. The computations are performed in a pipeline. Upon finishing the gate computations, the partial hidden and cell state arrays that were just computed for the input pair are ready to be used by the next input pair.

Unlike other solutions, the hidden state component is not computed for each input pair. Instead, a single hidden state array is maintained, which is used by all input pairs to compute the hidden state component of the forget, input, update, and output gates. For this reason, this can be considered a batch-stateful network[1], as it only stores the previous state of a whole batch. This enables the system to perform parallel computations for both the input pairs and the hidden state.

In order to save BRAM memory, only a single hidden and cell state array are maintained for the whole system. However, before being overwritten by the hidden and cell states of a new input pair, all hidden state values are used by the fully connected layer to compute their corresponding component on the output values, which are then, similarly to the computation results of the LSTM layer, stored in accumulators. Moreover, parallelisation is achieved, as each set of partial gate, hidden and cell state values of each input pair are computed in a pipeline.

It is also worth mentioning with more detail how the buffering and computation operations are performed at a higher level. First, both a batch of input pairs and a block of the weight matrices are traversed along their columns. Then, the next block of rows of the weight matrices is buffered, whereas the input pair batch is buffered again from the beginning. This means that the input

[1]Distinction between stateful and stateless networks is made in subsection 2.3.2.

pairs in a batch need to be rebuffered as many times as the number of individual block weight matrices, and it is not possible to reuse this data. Nevertheless, this appears to be the most sensible option. It not only enables the reuse of weights, but also the proper functioning of the system while storing only a fraction of the temporary accumulators (and, as a consequence, only a fraction of the BRAM) needed to compute the gates of the network (when the matrix is fully traversed along its columns, some components for computing the gates are fully calculated, and can immediately be used to compute the hidden state and the fully connected layer output).

4.1.3 Fixed-Point Design

The accelerator uses a fixed-point design for its internal computations. To perform this, the `ap_fixed.h` library provided by Vivado HLS was used.

The parameters W and I described in subsection 2.4.3 should be defined in accordance with the needs of the values to be fed into the accelerator. For this work, the parameter O is also used so that the values saturate when its maximum representation is exceeded. Despite the resulting increase in logic, this enables the circuit to avoid value overflow, which could completely alter the results in the circuit and eliminate their validity, making it possible to achieve good performance with smaller world bit-widths W.

Because the accelerator outputs the final value as an integer, the `ap_int.h` library provided by Vivado HLS was also used. Thus, it is possible to define only the bit length that is strictly needed to provide a valid output.

4.1.4 Coding Structure

The synthesised code contains the following types of modules:

- **Initialisation:** corresponds to the instantiation of all local variables (e.g. temporary variables, accumulators, buffers) that will be mapped to registers or BRAMs;

- **Buffering blocks:** correspond to all blocks that fetch data from DDR into the BRAM memory of the FPGA;

- **Computation blocks:** correspond to all blocks that perform computations (i.e. calculations related with the inputs x_t and the hidden state h_t, as well as the final computations for the output layer).

The blocks mentioned above are implemented in HLS as C++ functions, which work as a building block, and allow for a clear separation between the different modules. This separation also enables HLS to more easily check for data dependencies, and parallelise the execution of the different modules on the FPGA.

4.2 Constituent Modules

4.2.1 Initialisation

In Vivado HLS, BRAM is automatically used by explicitly declaring arrays in C++, so this mechanism was used to instantiate all arrays. Furthermore, single variables are stored in registers. Below follow the instantiated variables:

- **Hidden and cell state arrays, h and C**;
- **Accumulator for the outputs of the fully connected layer, L**: stores the output of the fully accumulated layer for all input pairs in a batch. It uses the *ARRAY_PARTITION* directive;
- **Buffer for input pairs, X:** stores a block of a batch of input pairs. Double-buffers[2] are used, and the *ARRAY_RESHAPE* directive is applied along the rows;
- **Buffers for the gate weight matrices corresponding to the input pairs, $\mathbf{W}_{if}$, $\mathbf{W}_{ii}$, $\mathbf{W}_{iz}$ and $\mathbf{W}_{io}$:** store a block of weight matrices used to compute the components of the input pairs. Double-buffers are used, and the *ARRAY_RESHAPE* directive is applied along the rows;
- **Buffers for the gate weight matrices corresponding to the hidden state, $\mathbf{W}_{hf}$, $\mathbf{W}_{hi}$, $\mathbf{W}_{hz}$, and $\mathbf{W}_{ho}$:** store a block of weight matrices used to compute the components of the input pairs. Double-buffers are used;
- **Accumulator matrices tmp_f_i, tmp_i_i, tmp_z_i, and tmp_o_i:** accumulate the input pair component of the values used to compute the gates. The *ARRAY_PARTITION* directive is applied along the columns;
- **Accumulator arrays tmp_f_h, tmp_i_h, tmp_z_h, and tmp_o_h:** accumulate the hidden state component of the values used to compute the gates.

As can be seen, array partition and reshaping occurs in a number of these variables. This was done to speed up the execution of the algorithm by taking full advantage of loop pipelining. Specifically, all variables needed for the computation of the partial results corresponding to the input pairs are either partitioned or reshaped, *in a cyclic manner*, with a *factor of 16*. This value was kept relatively low so as not to result in overall performance degradation (reshaping requires a greater output bit-width, which is achieved at the expense of a lower frequency of operation for the BRAMs). Furthermore, the dimension in which the pragmas are used was chosen considering the way the memory is accessed in the computational blocks in subsection 4.2.3.

4.2.2 Buffering Blocks

The following buffering blocks are instantiated by the accelerator:

[2] In Vivado HLS terminology, this is often called a *ping-pong* buffer.

- **Buffer_X_Batch:** inserts a batch of input pairs x_t into the buffer
- **Buffer_Wi_Blocks:** inserts a block of the $\mathbf{W}_{if}$, $\mathbf{W}_{ii}$, $\mathbf{W}_{iz}$, and $\mathbf{W}_{io}$ matrices into the buffer, which are used for computing the input pair component of the forget, input, update, and output gates
- **Buffer_Wh_Blocks:** inserts a block of the $\mathbf{W}_{if}$, $\mathbf{W}_{ii}$, $\mathbf{W}_{iz}$, and $\mathbf{W}_{io}$ matrices into the buffer, which are used for computing the hidden state component of the forget, input, update, and output gates
- **Init_tmp_Block:** starts the accumulators for both the input pair and hidden state components with the corresponding bias values
- **Init_L_Block:** starts the output value accumulator with the corresponding bias values

The aforementioned blocks are optimised for usage with an AXI Master interface. This protocol possesses the desired characteristics for buffering large amounts of data from external memory, as it is capable of performing sequential access bursts. According to [9], this requires any buffering block to use a *for loop*, and to use the pragma *PIPELINE* to allow for sequential bursts. Both optimisations have been implemented in all buffering blocks.

The *INLINE off* pragma was also used in all blocks. This enables Vivado HLS to synthesise each function as a single IP block, allowing for code optimisation and, most importantly, parallelising the execution of the blocks.

4.2.3 Computation Blocks

Below follows a description of all the building blocks used for computation. For simplification, the computation of the values for the forget f_t, input i_t, update z_t, and output o_t gates will be abbreviated to FIZO.

Similar to the buffering blocks, the *INLINE off* pragma was also used to enable parallelisation.

LSTM_Calc_Batch_X

Its execution is described by Algorithm 2. It computes the partial values of the FIZO gates corresponding to the input pairs x_t, and mainly consists of a module that fetches the buffered input pair values and weights associated with the input pairs. Summarily, it computes the multiplication between each input pair x_t and the weight matrices $\mathbf{W}_i$ for all gates in parallel.

Two HLS pragmas are used. *PIPELINE* enables pipelining for the second-level loop in order to speed up its execution, while also taking advantage of the reshaped and partitioned variables described in subsection 4.2.1. Using the *PIPELINE* directive implies, as discussed in subsection 2.4.2, that the loops below the pipeline are fully unrolled, which enables further parallelism, and eliminates the need of invoking the *UNROLL* pragma. *LOOP_FLATTEN* flattens the second-level loop for further scheduling optimisations.

Algorithm 2 LSTM_Calc_Batch_X

Variables: Input pairs matrix x
Weight matrices $\mathbf{W}_f$, $\mathbf{W}_i$, $\mathbf{W}_z$, $\mathbf{W}_o$
Accumulators *tmp_f*, *tmp_i*, *tmp_z*, *tmp_o*

for $i \leftarrow 1$ to BatchSize **do**
  for $j \leftarrow 1$ to BlockSize **do**
#pragma *HLS PIPELINE*
#pragma *HLS LOOP_FLATTEN*
    $x\_val \leftarrow x_{ij}$
    for $k \leftarrow 1$ to BlockSize **do**
      $tmp\_f_{ik} \mathrel{+}= x\_val \cdot \mathbf{W}_{f_{kj}}$
      $tmp\_i_{ik} \mathrel{+}= x\_val \cdot \mathbf{W}_{i_{kj}}$
      $tmp\_z_{ik} \mathrel{+}= x\_val \cdot \mathbf{W}_{z_{kj}}$
      $tmp\_o_{ik} \mathrel{+}= x\_val \cdot \mathbf{W}_{o_{kj}}$
    end for
  end for
end for

LSTM_Calc_Batch_h

Its execution is described by Algorithm 3. It computes the partial values of the FIZO gates corresponding to the hidden state h_t, and mainly consists of a module that accesses the hidden state values and weights associated it. Summarily, it computes the multiplication between the hidden state h_t and the weight matrices $\mathbf{W}_x$ for all gates in parallel.

Algorithm 3 LSTM_Calc_Batch_h

Variables: Hidden state h_i
Weight matrices $\mathbf{W}_{f_{kj}}$, $\mathbf{W}_{i_{kj}}$, $\mathbf{W}_{z_{kj}}$, $\mathbf{W}_{o_{kj}}$
Accumulators tmp_f_{ik}, tmp_i_{ik}, tmp_z_{ik}, tmp_o_{ik}

for $i \leftarrow 1$ to BlockSize **do**
  $h\_val \leftarrow h_i$
  for $j \leftarrow 1$ to BlockSize **do**
    $tmp\_f_{ik} \mathrel{+}= h\_val \cdot \mathbf{W}_{f_{kj}}$
    $tmp\_i_{ik} \mathrel{+}= h\_val \cdot \mathbf{W}_{i_{kj}}$
    $tmp\_z_{ik} \mathrel{+}= h\_val \cdot \mathbf{W}_{z_{kj}}$
    $tmp\_o_{ik} \mathrel{+}= h\_val \cdot \mathbf{W}_{o_{kj}}$
  end for
end for

In this case, no pragma is used. Although the *PIPELINE* pragma could be used for loop pipelining, it would require more resources with no increase in overall system performance, as this block does not create a bottleneck in the system.

LSTM_FIZO_Logistic_Batch

Its execution is described by Algorithm 4. It computes the final values of a block of FIZO gates, for each input pair x_t in the batch, and the new hidden h_t and cell C_t states. It first sums the input and hidden components for each component of the FIZO gates in parallel, and then uses these values to compute the corresponding hidden and cell states. Afterwards, a partial computation is performed for the output values of each input pair in the batch.

Algorithm 4 LSTM_FIZO_Logistic_Batch

Variables: Hidden state h_j
Cell state C_j
Accumulators $tmp\_f\_i_{ik}$, $tmp\_i\_i_{ik}$, $tmp\_z\_i_{ik}$, $tmp\_o\_i_{ik}$
Accumulators $tmp\_f\_h_{ik}$, $tmp\_i\_h_{ik}$, $tmp\_z\_h_{ik}$, $tmp\_o\_h_{ik}$
Weight matrix $\mathbf{W}_{l_{kj}}$
Outputs L_{ik}

#pragma *HLS ARRAY_PARTITION variable=h complete dim=1*
#pragma *HLS ARRAY_PARTITION variable=C complete dim=1*
for $i \leftarrow 1$ to BatchSize **do**
  for $j \leftarrow 1$ to BlockSize **do**
#pragma *HLS PIPELINE*
    $tmp\_h \leftarrow h_j$
    $tmp\_c \leftarrow C_j$

    $tmp\_f \leftarrow \sigma(tmp\_f\_i_{ij} + tmp\_f\_h_j)$
    $tmp\_i \leftarrow \sigma(tmp\_i\_i_{ij} + tmp\_i\_h_j)$
    $tmp\_z \leftarrow \tanh(tmp\_z\_i_{ij} + tmp\_z\_h_j)$
    $tmp\_o \leftarrow \sigma(tmp\_o\_i_{ij} + tmp\_o\_h_j)$

    $tmp\_c \leftarrow tmp\_f \cdot tmp\_c + tmp\_i \cdot tmp\_z$
    $tmp\_h \leftarrow tmp\_o \cdot \tanh(tmp\_c)$

    for $k \leftarrow 1$ to OutputSize **do**
      $L_{ik} \mathrel{+}= tmp\_h \cdot \mathbf{W}_{l_{kj}}$
    end for

    $h_j \leftarrow tmp\_h$
    $C_j \leftarrow tmp\_c$
  end for
end for

With this architecture, it was possible to minimise the amount of registers and BRAM blocks required by this block. This can be justified as the final values for every FIZO gate are calculated on-the-fly and then used to output the new values for the hidden and cell states. Moreover, the partial computations for the fully connected layer at the output are performed, which uses the values just computed for the hidden state, thus eliminating the need of storing them.

Moreover, a number of pragmas are used. *PIPELINE* enables pipelining for the second-level loop, with the purpose of performing multiple operations over different hidden state values for the same input pair. Furthermore, just as in **LSTM_Calc_Batch_X**, this implies that the loops below the pipelined loop are fully unrolled. *ARRAY_PARTITION* is then used to partition the hidden and cell state arrays[3] completely, so that each array value is instead stored in a register for increased performance.

LSTM_Logistic_Calc

Serves as a wrapper for the *max* function, which computes, for each input pair x_t, the index of the maximum value of the output array. It returns the index of the output value with maximum probability.

4.2.4 Auxiliary Blocks

4.2.4.1 Gaussian Pseudo-Random Number Generator

Because both the hidden state h_t and the cell state C_t arrays need to be randomly generated when the LSTM is initialised, it was necessary to implement a module that generated random values. Specifically, networks should preferably be initialised using Gaussian random values.

One immediate solution consists of generating two random numbers following a uniform distribution and then perform a Box-Müller transformation [27]. However, this would require a large amount of resources on fabric, and was thus discarded.

As a solution, a Gaussian pseudo-random number generator (GPRNG) using linear feedback shift registers (LFSRs) was instantiated. The chosen solution consists of the 4-LFSR GPRNG presented in [28], as it guarantees adequate statistical properties. Its operation principle rests on the *central limit theorem*, according to which a Gaussian distribution can be obtained through the addition of a sufficiently large number of uniformly distributed random values.

The 4-LFSR GPRNG is depicted in Figure 4.3. It uses four Fibonacci LFSR instances to implement the PRNGs, which generate new values each clock cycle. For feeding the LFSRs, a maximum-length polynomial of order 16 was chosen: $x^{15}+x^{14}+x^{12}+x^3$. Each LFSR is initialised by a unique seed that is sent as an input to the module. The values generated by the LFSRs are then put into a two-level tree in order to transform the uniformly-distributed numbers, by means of sums and bit shifts, into a single number with a Gaussian distribution.

[3] The hidden and cell state arrays used by this function correspond to a partial, buffered copy of the hidden and cell state arrays in the global system. The buffering and writing operations for those global arrays were omitted for simplicity.

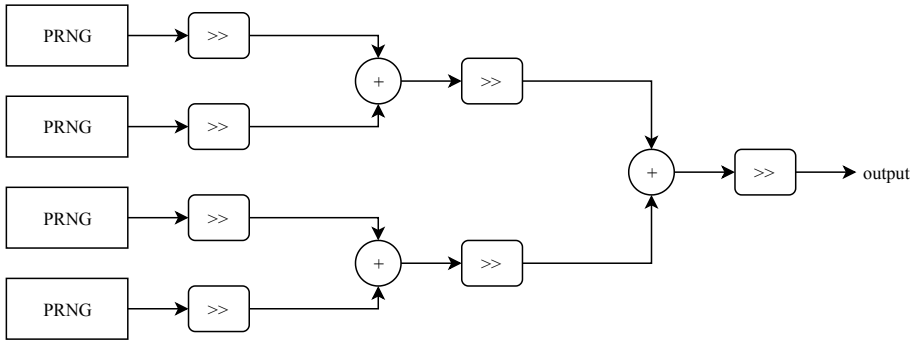


Figure 4.3: Architecture of the 4-LFSR GPRNG [29]

4.2.4.2 Activation functions

In compliance with the aforementioned LSTM formulas in Equation 2.7, the sigmoid and hyperbolic tangent functions were implemented. Their implementation on hardware can be somehow convoluted due to the use of exponentiation and division[4].

There are a number of possible solutions for this. Vivado HLS provides a CORDIC library which implements the hyperbolic tangent function, but this is an inefficient solution because it was designed to perform well for sequential calculations (where each computed value differs from the previous one by a small delta), whereas in this case the values differ arbitrarily from each other. It is possible to use a look-up table, however, because a large number of input and output value pairs is required, it would be expensive in terms of memory. Polynomial approximation is also an option, but it may not be well translated to fixed-point notation.

With both performance and resource utilisation in mind, the adopted solution consists of a piecewise linear approximation (PLAN) implementation [30], which has been recognised to be an efficient solution for FPGAs [31]. Its architecture is depicted in Figure 4.4. This algorithm first checks the absolute value of the input, and then performs a bit-wise shift operation followed by a bias term. The final result is then computed depending on the sign of the input, thereby taking advantage of the symmetry of the function.

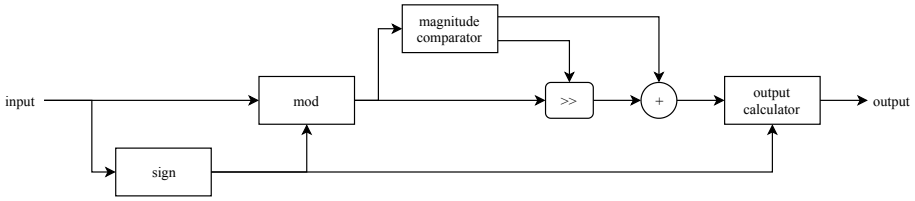


Figure 4.4: Architecture of the PLAN algorithm

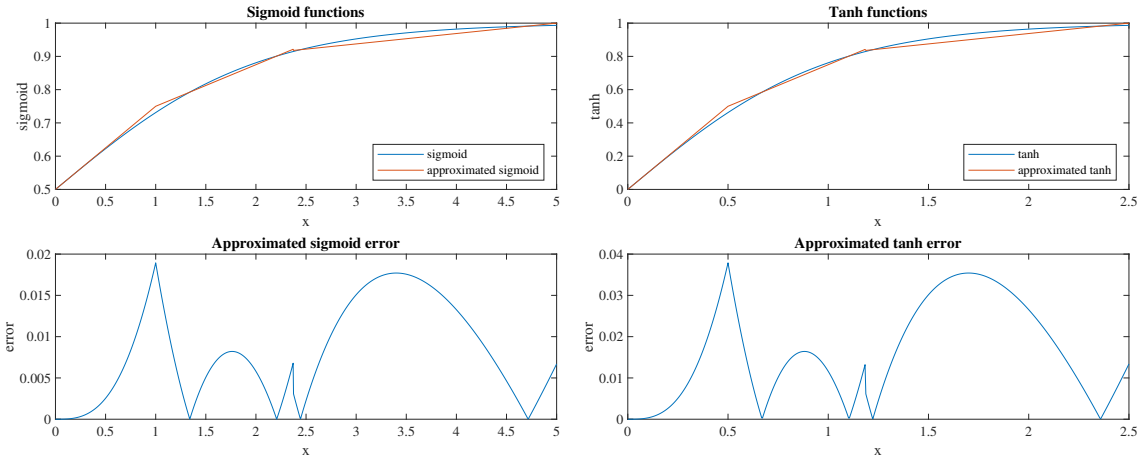
Although PLAN was originally conceived for the sigmoid function, a similar procedure can be used to implement the hyperbolic tangent, as it can be viewed as a rescaled sigmoid, as shown

[4] See Equation 2.3-2.4.

in Equation 4.1.

$$\tanh(x) = 2 \cdot \sigma(2x) - 1 \tag{4.1}$$

Figure 4.5 shows the simulated results in Matlab.



(a) sigmoid function (b) hyperbolic tangent function

Figure 4.5: Comparison of the original functions with PLAN, considering a decimal precision of 16 bits, and its corresponding error (due to symmetry, only positive values are shown)

From the results, it can be concluded that the waveforms and the corresponding error are identical for both functions, with the error in the hyperbolic tangent being exactly twice as large in magnitude due to the rescaling factor in Equation 4.1. Both functions report a tracking error below 2% in comparison with the signal excursion, while showing an overall smooth behaviour with few discontinuities, thus proving PLAN to be a good choice for implementation.

4.3 Summary

In this chapter, the proposed accelerator architecture was described. The overview in Section 4.1 presented the most important aspects of the accelerator, namely its description, the block-batching architecture used for matrix multiplication, the fixed-point design used for internal computations, and the coding structure used. The constituent modules were explored in Section 4.2, where it was possible to further explore both the arrangement of the variables in memory and the optimisations performed during the computation of the output values of the LSTM.

Chapter 5

Results

The design presented in Chapter 4 was synthesised using Vivado HLS 2019.1, and implemented on a board model using Vivado 2019.1. Section 5.1 starts by explaining how the architecture was validated during simulation. Section 5.2 explores the synthesis procedures and performs a design-space exploration for a combination of batch and block sizes. Section 5.3 presents the accuracy results that can be obtained with the system using the MNIST dataset, with a focus on its variation for different word bit-widths for its data types. Section 5.4 describes the implementation performed on the board model.

5.1 Simulation Validation

The simulation validation of the accelerator was performed in two separate steps: C Simulation and C/RTL Cosimulation.

First, the C Simulation was performed in Vivado HLS, which involved the creation of a C++ testbench file. This file loaded the input pair values and the weight matrices from binary files into local arrays, which were then accessed (and buffered) by the accelerator. This file also compared the inferred results against the expected results stored in main memory. Because of its ability of obtaining results identical to those of C/RTL Co-simulation in a fraction of the time, it was used for obtaining the results observed in Section 5.3.

After this, C/RTL Cosimulation was performed. This simulation also uses the C++ testbench to load the necessary values. However, the simulation now uses a circuit that was previously translated from C++ to HDL, which is then used to generate a RTL circuit. Thus, this simulation is cycle-accurate and provides a close representation of the final behaviour of the accelerator on the board model. This means that the function that was previously simulated in C++ is now simulated using RTL. A screenshot depicting some of the registered waveforms is presented in Appendix C.

5.2 Synthesis Results

The proposed network was synthesised for a Xilinx Virtex-7 XC7VX485T FPGA. For assessing the impact of parameter tuning on the performance of the block-batching technique, which was discussed in subsection 4.1.2, a network was successively synthesised for different batch, S_{batch}, and block, S_{block}, sizes.

The network parameters used for this network are presented in Table 5.1. The hidden dimension was chosen to enable comparison between a wider range of S_{block} values, whereas the number of samples and the input dimension were chosen to suit the needs of the MNIST dataset [32] that will be used in Section 5.3.

nSamples	InputDim	HiddenDim	OutputDim
10000	784	128	10

(a) Network dimensions

Input	Hidden	Mem	Calc	Out
<18,2>	<14,6>	<14,6>	<14,6>	<4>

(b) Word and integer-part <W,I> bit-widths used (Out only uses <W>)

Table 5.1: Network parameters used for studying the effects of block-batching

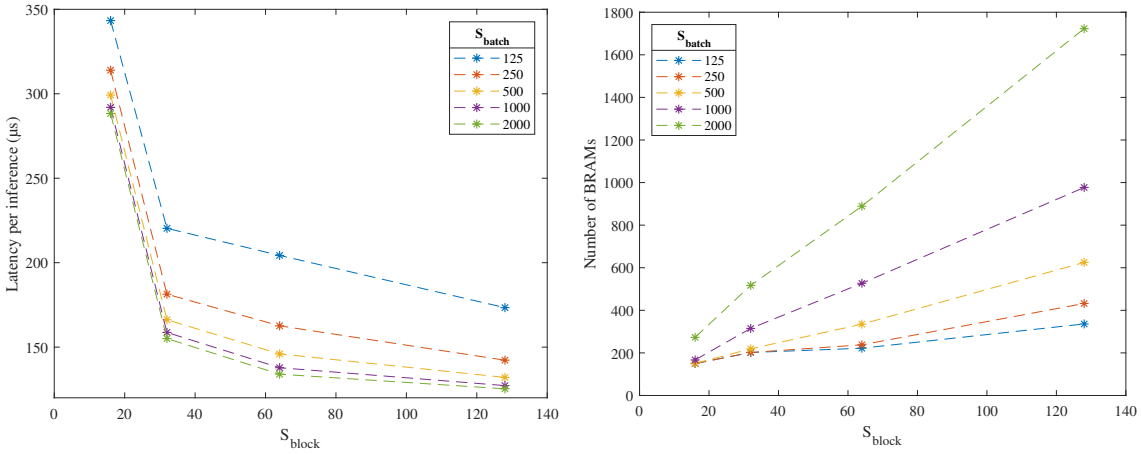
To better understand the impact of varying S_{batch} and S_{block} in performance, latency per inference was calculated for a number of $\{S_{batch}, S_{block}\}$ pairs, with larger latency values meaning worse performance. It is shown in Equation 5.1, and is obtained by dividing the latency (in clock cycles) of the accelerator by the product of the number of processed samples with the frequency.

$$\text{latency per inference} = \frac{\text{latency}}{\text{nSamples} \cdot f} \tag{5.1}$$

The results were obtained from the estimates provided after synthesis in Vivado HLS. For the circuits tested, it was verified that the synthesis estimates were similar to the results obtained after C/RTL Cosimulation. The values obtained are presented in Figure 5.1. The corresponding resource utilisation and frequency values for all synthesised architectures are reported in Appendix B.

It is concluded that increasing both S_{batch} and S_{block} will result in performance improvements. In addition, resource utilisation, apart from BRAM usage, is virtually independent from S_{batch}. It was also verified that the estimated frequency ($f = 114.29\,\text{MHz}$) was equal for all synthesised architectures. Resource utilisation for different S_{block} values is presented in Table 5.2.

Significant improvements occur when increasing S_{block} from 16 to 32. This happens because of the arrays storing the temporary accumulators during input pair multiplications. These are partitioned with a factor of 16 (see subsection 4.2.1) to dual-port BRAM memories. Because of this, for $S_{block} = 16$, only one of the ports of each partitioned array is used for reading data, whereas for $S_{block} = 32$ (and for any value above that) both ports are used. This explains the



(a) variation in latency per inference

(b) variation in number of BRAMs

Figure 5.1: Impact of S_{batch} and S_{block} variation in latency and BRAM usage on the simulated accelerator

S_{block}	DSP48E		FF		LUT	
16	83	(2.96%)	15048	(2.48%)	78037	(25.70%)
32	147	(5.25%)	18663	(3.07%)	88405	(29.12%)
64	275	(9.82%)	23860	(3.93%)	104608	(34.46%)
128	531	(18.96%)	42638	(7.02%)	135145	(44.51%)

Table 5.2: Resource utilisation results for different S_{block} values ($S_{batch} = 500$) for the simulated accelerator

diminishing performance improvements for larger S_{block} values, which occur due to to the addition of more levels of pipeline on the LSTM_Calc_Batch_X module. The minimum IIs of the pipelines achieved by Vivado HLS for $S_{block} = \{16, 32, 64, 128\}$ are $\{3, 3, 6, 12\}$, respectively.

A number of improvements also occur when changing S_{batch} from 125 to 250, while using the same BRAM resources. This likely happens because the amount of data buffered by the system is insufficient to fill the BRAMs (thus not fully using those resources), while increasing the number of accesses to off-chip memory. Notable improvements still occur for $S_{batch} = 500$.

If the need of choosing a balanced solution arises (e.g. the accelerator needs to be synthesised alongside other IP blocks in the fabric), for all values studied, the architecture with $\{S_{batch}, S_{block}\} = \{500, 64\}$ appears to be best option.

Additional Considerations

Tests were also performed using a hidden dimension of 256 while keeping the remaining parameters of Table 5.1 equal. It was observed that further increasing S_{block} from 128 to 256 actually resulted in performance degradation, likely due to the additional intermediate computations.

During experimentation, it was also observed that unrolling either the pipeline-level loop or the top-level loop of the LSTM_Calc_Batch_X module, unlike expected, would result in a performance decrease. The lack of performance improvements occurs due to the bottleneck on memory

access (with $S_{block} = 32$, all memory ports are busy), whereas the increase in latency is likely due to more complex routing.

5.3 Accuracy Measurements

5.3.1 Overview

Accuracy measurements were performed over the MNIST dataset. It consists of a database of images of individual handwritten digits with resolution of 28×28. The values of each pixel consist of a value in the interval $[0, 1]$. To process the data using an LSTM network, the digits were reshaped from a 28×28 matrix to an array of size 784. The arrays were then fed as inputs. The hidden state of the network can be set to an arbitrary number, however it's important to note that a larger dimension will result in more components of the forget, input, update, and output layers, which retain more information (thus requiring more off-chip memory for storage) for correct prediction. The output dimension of the fully connected layer is then set to 10, which corresponds to the number of digits to be recognised. After passing through a *max* function, which captures the index of the output array for each input pair with the largest value, the network outputs the predicted number.

The LSTM network was trained using an existing software implementation. This led to the choice of PyTorch, as it provides flexibility for creating the networks using a front-end interface in Python, a high-level programming language easy to understand, with a back-end implementation optimised for high performance (C++ with SIMD instructions).

5.3.2 Training

Training of the network for MNIST was performed using PyTorch, with the code used for training in Appendix A. With this procedure, it is possible to generate a network with an arbitrary hidden dimension, which defines the granularity (i.e. the number of weights) of the network. The number of epochs used for training can also be defined, and determines how many passes are performed through the training set. For accuracy purposes, it is relevant to choose an appropriate number of epochs: using a small value may result in a network with not enough training (which will provide poor inference results), whereas a large value will result in a network that is overfitting the training set (which results in a less robust network that performs worse during real-world usage).

5.3.3 Results

Accuracy measurements were performed for the 10000 elements of the test set of the MNIST dataset. The network parameters of the LSTM used for this purpose are identical to those of Table 5.1a, but with the hidden dimension set to 32.

The purpose of these measurements is to understand the impact of changing the word and integer-part bit-width <W,I> pairs of the variables (except for the input pairs, which are kept

constant at <18,2>) on the accuracy for different training epochs. The results are presented in Figure 5.2 and Table 5.3, with PyTorch being used for benchmarking.

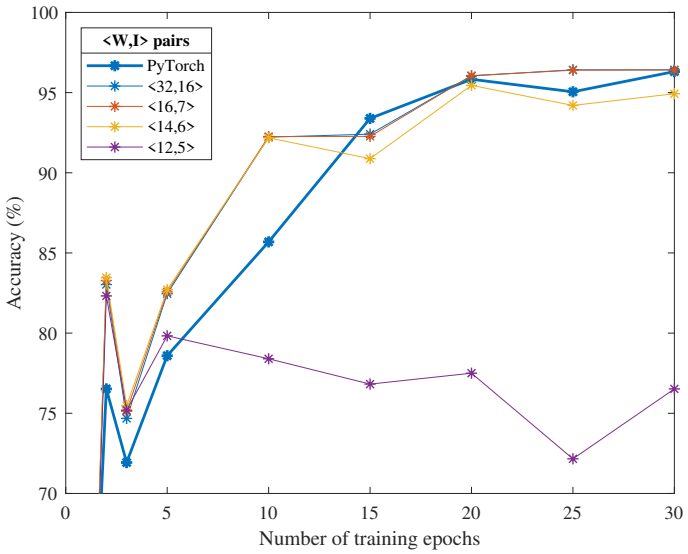


Figure 5.2: Network accuracy for the MNIST training set with respect to the number of training epochs and the <W,I> pairs

Epochs	**1**	**2**	**3**	**5**	**10**	**15**	**20**	**25**	**30**
PyTorch	49.52%	76.52%	71.93%	78.59%	85.69%	93.39%	95.83%	95.05%	96.31%
<32,16>	44.29%	83.03%	74.68%	82.46%	92.22%	92.41%	96.04%	96.42%	96.42%
<16,7>	44.65%	83.28%	75.13%	82.57%	92.26%	92.27%	96.06%	96.4%	96.41%
<14,6>	44.97%	83.46%	75.48%	82.71%	92.18%	90.88%	95.45%	94.2%	94.93%
<12,5>	44.3%	82.32%	75.19%	79.83%	78.4%	76.82%	77.5%	72.16%	76.52%

Table 5.3: Network accuracy for the MNIST training set with respect to the number of training epochs and the <W,I> pairs

It can be concluded that, for the majority of training epochs, the results on the accelerator surpass those of PyTorch. The pairs <32,16> and <16,7> show the best performance, whereas the pair <14,6> registers a maximum degradation of accuracy of 2.2% with 1.14x less memory resources. Thus, it can be used to obtain reliable results where low latency is required, or where memory resources are limited. The pair <12,5> shows a marked performance drop and is not a viable option.

Further testing was performed with the pairs <16,7> and <14,6> while varying the word and integer-part bit-widths for input, in an effort to further reduce on-chip memory requirements. For the purpose, a network with a hidden dimension of 32 and 30 training epochs was used. The results are presented in Figure 5.3.

It is observed that it is also possible to further reduce on-chip memory requirements by 1.29x in relation to the original bit-width (or by 1.14x in comparison with [18, 20, 26]) by reducing

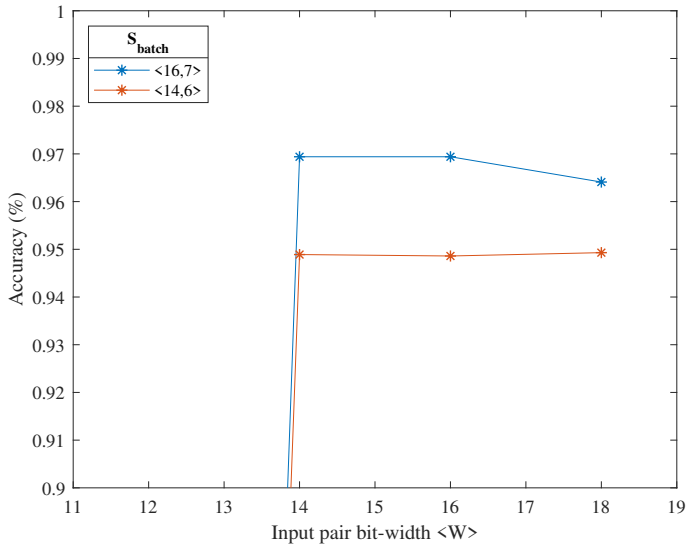


Figure 5.3: Network accuracy for the MNIST training set with respect to the word bit-width <W> of the input pairs (I = 2)

the bit-width to 14 bits without incurring performance reduction. When tested with a word bit-width of 12 bits, it was no longer possible to obtain adequate results, resulting in the abrupt drop observed in Figure 5.3.

Additionally, thanks to its flexibility, this architecture enables increasing the word bit-width W, which helps to improve performance in networks where larger bit-widths are necessary to obtain good performance results. This feature is not present in the solutions presented in Table 3.1, which are limited to a (usually fixed) maximum word bit-width of 16 bits and that, to our best understanding, do not allow to tune the integer-part bit-width.

5.4 Board Implementation

5.4.1 Overview

The proposed accelerator was implemented on a board model. The implementation was performed on a Xilinx Virtex-7 VC707 Evaluation Board, which contains the Xilinx Virtex-7 XC7VX485T FPGA that was used as simulation target in Section 5.2.

Vivado 2019.1 was used for creating the block design that implemented the circuit on fabric, and its diagram is depicted in Figure 5.4. The most relevant components of the design are described as follows:

- **Microblaze:** consists of a soft microprocessor core, implemented on reconfigurable fabric, that is used to start the accelerator by loading the necessary drivers and the base memory addresses for accessing the the input pair values, weight matrices, and output values;
- **Memory Interface Generator (MIG):** provides the memory interface from the Microblaze and the accelerator to DDR memory;

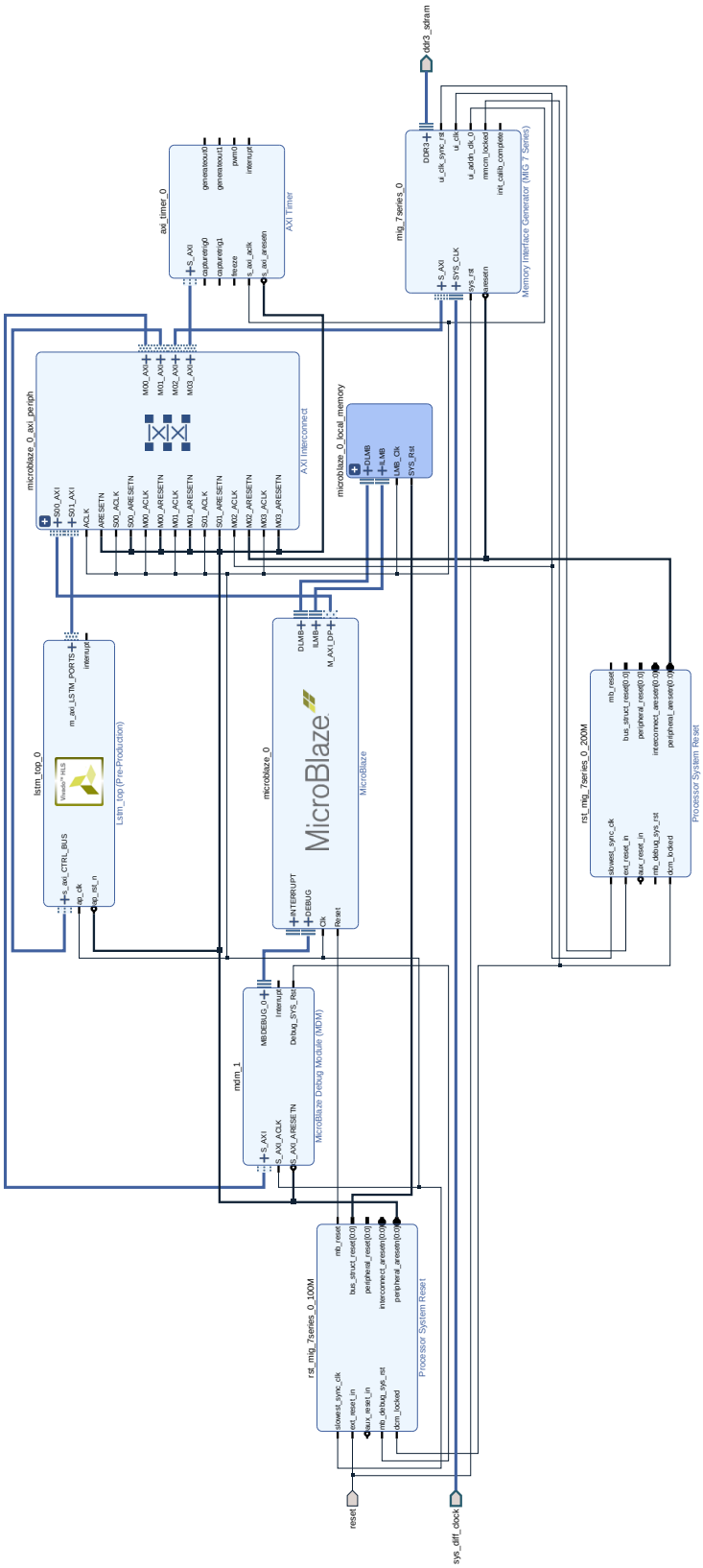


Figure 5.4: Block design for implementing the proposed accelerator on the VC707 board

- **Lstm_top:** consists of the accelerator module, which interacts with the Microblaze for initialisation, and with the MIG for performing memory requests directly to DDR (i.e. without processor intervention);
- **AXI Timer:** a built-in timer used for performance monitoring;
- **AXI Interconnect:** an interconnect that performs all the connections between the different components in the circuit.

After design validation, synthesis and implementation were executed, and the bitstream (i.e. the data to be sent to the FPGA to generate the design blocks) was generated.

Xilinx SDK was used to verify the behaviour of the accelerator. For this purpose, a testbench similar to the one used for C Simulation was created. This code compared all output values against their references, and measured the execution time using the AXI Timer. The input pairs and weight matrices were passed from a computer to DDR via the JTAG connection. For the tested networks, it was verified that the accuracy results acquired on-chip were identical to those registered in simulation.

5.4.2 Results

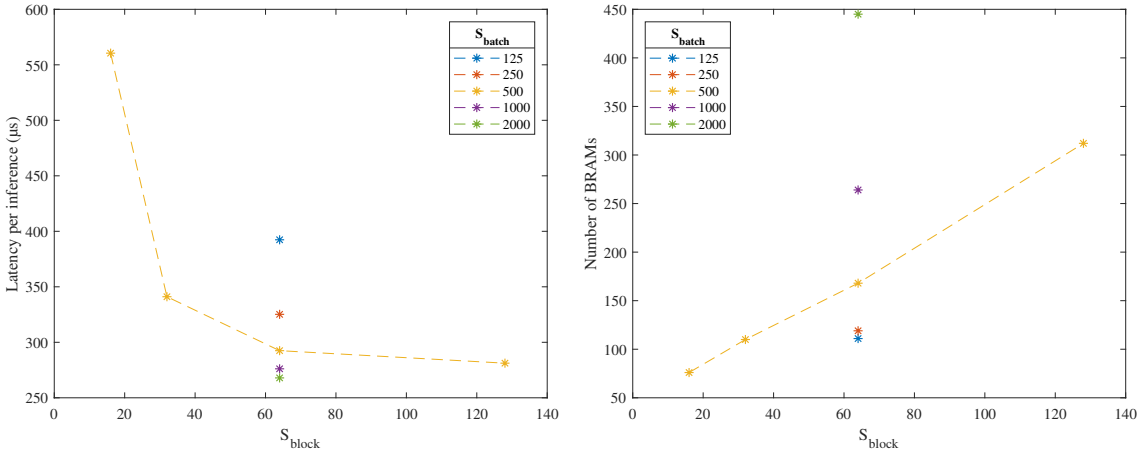
Similar to the procedure described in Section 5.2, the proposed network was implemented for a number of $\{S_{batch}, S_{block}\}$ pairs. The network parameters are those presented in Table 5.1 to enable a comparison with the estimated values from synthesis. The latency results correspond to values measured on-chip by the AXI Timer, whereas the resource utilisation values are fetched from Vivado, and correspond exclusively to the IP block containing the LSTM accelerator, as seen in Figure 5.4.

The results obtained are presented in Figure 5.5, with the resource utilisation and frequency values for all synthesised $\{S_{batch}, S_{block}\}$ pairs reported in Appendix B. Resource utilisation for different S_{block} values is presented in Table 5.4.

For the circuits tested, it was verified that resource utilisation and latency per inference exhibit similar evolution patterns regarding those of the simulated circuit, namely for the evolution of latency per inference and BRAM usage, for different S_{batch} values, with respect to S_{block}. As expected, resource utilisation, apart from BRAM usage, is again virtually independent from S_{batch}.

However, it was verified that the actual resource utilisation and latency per inference values obtained from simulation in Vivado HLS are noticeably different from those obtained from the board implementation. The number of BRAMs used is reduced by 50%, and resource utilisation for the remaining parameters is also heavily reduced (e.g. LUT utilisation is reduced by up to 80%). These optimisations are likely a result of a number of optimisations performed by Vivado during the synthesis, placement, and routing stages.

Additionally, the implemented solution appears to be between 1.68x to 1.87x slower in comparison with the simulated accelerator. This is due to a number of reasons. Namely, the usage of physical DDR memory results in additional delays, which in turn are dependent on how memory



(a) variation in latency per inference

(b) variation in number of BRAMs

Figure 5.5: Impact of S_{batch} and S_{block} variation in latency and BRAM usage on the board implementation

S_{block}	DSP48E		FF		LUT	
16	83	(2.96%)	9654	(1.59%)	16197	(5.33%)
32	147	(5.25%)	10044	(1.65%)	19305	(6.36%)
64	275	(9.82%)	12100	(1.99%)	23834	(7.85%)
128	531	(18.96%)	23857	(3.93%)	31462	(10.36%)

Table 5.4: Resource utilisation results for different S_{block} values ($S_{batch} = 500$) for the board implementation

is accessed. To the best of our knowledge, accesses are performed using AXI Master and accessing the arrays in burst mode, thereby limiting memory access penalties. Furthermore, the physical interconnects in the board also result in increased transfer times.

5.4.3 Comparison

5.4.3.1 Other Platforms

The proposed network was compared against other hardware platforms. For this purpose, an LSTM network with the parameters presented in Table 5.1a was implemented on both a desktop and an embedded CPU, which should be used for similar applications. For the former, an Intel Core i7-3770 CPU was used, which executed the code in PyTorch mentioned in Section 5.3 on a single thread. For the latter, a dual-core ARM Cortex-A9 CPU of the Zynq-7000 SoC was used, which executed the C++ code used to generate the circuit on the FPGA on one of its cores. With the purpose of performing a fair comparison on the ARM CPU, all buffering operations were removed, and the compiler flags were set to full optimisation, with automatic usage of SIMD NEON vectorisation instructions[1]. The reported latency times do not consider the initial data

[1]The compiler flags `-O3` and `-mcpu=cortex-a9 -mfpu=neon -ftree-vectorize-mvectorize-with-neon-quad` were used for the purpose.

transfer to DDR memory. The results are presented in Table 5.5.

Platform	Desktop	Embedded	Proposed
Processor	Intel Core i7-3770	ARM Cortex-A9	Xilinx Virtex-7 XC7VX485T
Frequency	3.40 GHz	667 MHz	100 MHz
Technology	22 nm	28 nm	28 nm
Memory	8 GB DDR3	512 MB DDR3	1 GB DDR3
Precision	64-bit floating-point	32-bit floating-point	14-bit fixed-point
Latency per inference	582.54 µs	3080.96 µs	292.47 µs

Table 5.5: Performance comparison against CPU implementations

Compared with the Intel and ARM processors, the proposed solution provides a speed-up of 1.99x and 10.53x, respectively, while providing similar accuracy results.

5.4.3.2 Other Works

A comparison against previous works is shown in Table 5.6. The latency per inference values were normalised across the different solutions by taking into consideration variations in frequency and compression ratios.

Paper	Wang [18]	Cao [20]	Wang [22]	Rybalkin [24]	Que [26]	Proposed
Device	Xilinx Virtex-7 XC7VX690T	Intel Arria 10 GX1150	Intel Arria 10 SX660	Xilinx Zynq XCZU7EV	Xilinx Zynq XC7Z045	Xilinx Virtex-7 XC7VX485T
Frequency (MHz)	200	200	200	266	142	100
Memory storage	on-chip	off-chip	off-chip	on-chip	off-chip	off-chip
Data representation	16-bit fixed-point	16-bit fixed-point	8-bit fixed-point	1-to-8-bit fixed-point	16-bit fixed-point	14-bit fixed-point
Compression ratio	7.9	8	8	-	-	-
Network dimensions*	39, 512, N/A	153, 1500, 512	153, 1024, N/A	1024, 128, 82	2048, 256, N/A	784, 128, 10
Latency per inference (normalised)	9.8 µs (154.84 µs)	2.4 µs (38.4 µs)	23.9 µs (382.4 µs)	-	610 µs (866.2 µs)	292.47 µs (292.47 µs)

* Indicates the input, hidden, and output dimensions. N/A indicates that no fully-connected layer is implemented

Table 5.6: Comparison against previous works

Concerning the performance results, it is concluded that the proposed accelerator provides a performance in line with the aforementioned works, when considering the normalised latency per inference values. Namely, only the works from Wang [18] and Cao [20] report lower values.

Nevertheless, it is important to point out that any comparison should be performed carefully due to the several differences between implementations. For example, it is expected that solutions making exclusive use of on-chip memory perform faster due to lower latency. On the other hand, network dimensions, which heavily influence latency results, are different for each implementation. Board frequency and compression ratios will also influence the results (despite this, it was possible to normalise the values regarding these parameters). Additionally, the results will depend on the board in which the accelerator is implemented, and on the technology (i.e. HDL or HLS) that describes the circuit.

Chapter 6

Conclusions

This work focused on the implementation of an LSTM network on FPGA using HLS tools. By making use of these tools, it was possible to produce a parametrisable architecture which enables the effortless specification of the network to be implemented. This also allows synthesising LSTM accelerators with different network characteristics by modifying a small number of constants in the source code.

The accelerator implements a block-batching architecture that, together with double-buffering, is intended to balance the flexibility of using off-chip memory for storage with the advantage of on-chip memory for fast access during computations. Furthermore, the hidden state is computed on a per-batch basis, therefore using a batch-stateful LSTM architecture, which unloads the system from performing additional operations and enabling the computation of the hidden state in parallel with computations for the input pair. In order to achieve a higher level of parallelism and the reuse of the elements instantiated on fabric, the implementation of the most computationally-intensive blocks is performed using pipelines.

Unlike previous works, where the word bit-width is often limited to 16 bits, the proposed accelerator enables tuning the word bit-width and the integer part bit-width of its internal variables. This opens up the possibility of using the accelerator for networks that either require higher precision, or lower precision (i.e. in scenarios where lower precision does not incur in major performance degradation), while keeping the accelerator optimised for the bit-width required. For the MNIST dataset, it was possible to reduce on-chip memory requirements for both the input pairs and the weight matrices of up to 1.14x by using the aforementioned strategy, in comparison with previous works.

Additionally, the proposed accelerator was implemented on a board model, and further performance measurements were performed. Speed-ups of 1.99x and 10.53x was registered in comparison with a desktop and an embedded CPU, respectively. The registered performance was in line with that of previous works.

Future Work

Further work can be carried in a number of directions. For one, it is possible to further decrease on-chip memory requirements by exploring pruning techniques for the weight matrices. Additionally, it is also possible to improve performance by performing pre-processing on datasets, for instance by clamping values below a threshold to zero, or by skipping multiplication of null input pair values altogether. Nevertheless, both of these approaches will likely reduce the flexibility of the accelerator in question, as well as requiring modifications to its architecture.

Besides this, additional work can be performed on network binarisation, which consists of using binary values instead of fixed-point weights. This can be highly efficient on hardware, as it could enable the use of shift-based operations, thereby eliminating the need of more complex matrix multiplication operations. This strategy is likely dependent on the characteristics of the dataset to be inferred.

Appendix A

Network Training in PyTorch

```
import argparse
import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim
from torchvision import datasets, transforms

# Outputs variables to files with fixed-point precision
def export_weights(tensor, name, db, netmode, decimal_bits, mode = 'default'):
  # Defines exponent for desired precision
  precision = (2 ** decimal_bits)

  # Performs default treatment
  if (mode == 'default'):
      (tensor.detach().numpy() * precision).astype('<u4').tofile("../../model/"
          + db + "/" + netmode + "/bin/" + str(num_epochs) + "/" + str(
         decimal_bits) + "/" + name + ".bin")
      with open("../../model/" + db + "/" + netmode + "/csv/" + name + ".csv",
          "w") as f:
        np.savetxt(fname = f, X = tensor.detach().numpy(), delimiter = ';')

  # Splits LSTM matrices into their specific weights
  elif (mode == 'split'):
    # LSTM matrices use a special splitting treatment
    if (name.find('lstm') == 0):

      # Chunks tensor and attribute appropriate naming
      num_chunks = 4
      tensor_chunked = tensor.chunk(num_chunks)
      name_ih = ['ii', 'if', 'iz', 'io']
      name_hh = ['hi', 'hf', 'hz', 'ho']

      # Prints each chunk in a separate file
```

```
31        for i in range(num_chunks):
32          if (name.find('ih') != -1):
33            name_weight = name.replace('ih', name_ih[i])
34          elif (name.find('hh') != -1):
35            name_weight = name.replace('hh', name_hh[i])
36          (tensor_chunked[i].detach().numpy() * precision).astype('<u4').tofile("
                ../../model/" + db + "/" + netmode + "/bin/" + str(num_epochs) + "/
                " + str(decimal_bits) + "/" + name_weight + ".bin")
37
38      # Other weights are treated as usual
39      else:
40        (tensor.detach().numpy() * precision).astype('<u4').tofile("../../model/"
              + db + "/" + netmode + "/bin/" + str(num_epochs) + "/" + str(
              decimal_bits) + "/" + name + ".bin")
41
42  # Defines LSTM network as class
43  class LSTMNet(nn.Module):
44
45    def __init__(self, input_size, hidden_size, num_layers, output_size,
          batch_size = 1):
46      super(LSTMNet, self).__init__()
47
48      # Initialise LSTM inner variables
49      self.input_size   = input_size
50      self.hidden_size  = hidden_size
51      self.num_layers   = num_layers
52      self.batch_size   = batch_size
53      self.output_size  = output_size
54
55      # Define the LSTM layer
56      self.lstm = nn.LSTM(self.input_size, self.hidden_size, self.num_layers,
            batch_first = True)
57
58      # Define the output layer
59      self.linear = nn.Linear(self.hidden_size, self.output_size)
60
61    def init_states(self):
62      # Initialises hidden state (w/ batch_size = 1)
63      self.h0 = torch.randn(self.num_layers, self.batch_size, self.hidden_size).
            requires_grad_()
64
65      # Initialises hidden state (w/ batch_size = 1)
66      self.c0 = torch.randn(self.num_layers, self.batch_size, self.hidden_size).
            requires_grad_()
67
68    def forward(self, x):
69      # Performs forward pass
70      # shape of out: [input_size, batch_size, hidden_size]
```

```
71      # shape of (hn, cn): (a, b), where a and b both have shape (num_layers,
            batch_size, hidden_size).
72      out, (hn, cn) = self.lstm(x, (self.h0.detach(), self.c0.detach()))
73
74      # Fetches the output from the final timestep
75      # Can pass out to the next layer if it is a seq-to-seq prediction # (CHECK
            WHAT THIS IS!!)
76      yh = self.linear(out[:, -1, :])
77      return yh
78
79  # Parse inputs
80  # Training settings
81  parser = argparse.ArgumentParser(description='LSTM application in MNIST
        database')
82  parser.add_argument('--batch-size', type=int, default=1, metavar='N',
83                      help='input batch size for training (default: 1)')
84  parser.add_argument('--decimal-bits', type=int, default=8, metavar='N',
85                      help='number of decimal bits to be exported (default: 8)')
86  parser.add_argument('--hidden-size', type=int, default=32, metavar='N',
87                      help='size of hidden layer (default: 32)')
88  parser.add_argument('--learning-rate', type=float, default=0.1, metavar='LR',
89                      help='learning rate (default: 0.1)')
90  parser.add_argument('--num-epochs', type=int, default=5, metavar='N',
91                      help='number of epochs for training (default: 5)')
92  parser.add_argument('--num-layers', type=int, default=1, metavar='N',
93                      help='number of layers in the LSTM (default: 1)')
94
95  args = parser.parse_args()
96
97  # Initialises LSTM variables
98  input_size  = 28 * 28
99  hidden_size = args.hidden_size
100 num_layers  = args.num_layers
101 output_size = 10
102
103 # Initialises helper variables
104 learning_rate = args.learning_rate
105 batch_size    = args.batch_size
106 decimal_bits  = args.decimal_bits
107 num_epochs    = args.num_epochs
108
109 # Loads train and test sets
110 train = torch.utils.data.DataLoader(
111                     datasets.MNIST('data',
112                     train=True,
113                     download=True,
114                     transform=transforms.ToTensor()),
115                     batch_size=batch_size,
116                     shuffle=True)
```

```
117 test  = torch.utils.data.DataLoader(
118                     datasets.MNIST('data',
119                     train=False,
120                     download=True,
121                     transform=transforms.ToTensor()),
122                     batch_size=batch_size,
123                     shuffle=True)
124
125 # Defines model properties
126 model = LSTMNet(input_size, hidden_size, num_layers, output_size)
127
128 # Prints model and parameters
129 print("Model overview:")
130 print(model)
131 print("Model state_dict:")
132 for param_tensor in model.state_dict():
133     print(param_tensor, "\t", model.state_dict()[param_tensor].size())
134
135 # Uses mean-squares error as loss criterion
136 criterion = nn.CrossEntropyLoss()
137
138 # Uses stochastic gradient descent as optimisation function
139 optim = optim.SGD(model.parameters(), lr = learning_rate)
140
141 # Initialises value for storing histogram
142 # hist = np.zeros(num_epochs)
143
144 # Prints information message for starting retraining
145 print("Starting training...")
146
147 # Trains the model (using a **batch** method)
148 for epoch in range(num_epochs):
149   # Initialises hidden state (initialise before model(x) if LSTM is to be
        stateless)
150   model.init_states()
151
152   # Sets model to training mode
153   model.train()
154   # Performs computations using mini-batch (batch_size = 1)
155   for i, (x, y) in enumerate(train):
156     # Changes shape of x and enables gradient accumulation
157     x = x.view(batch_size, 1, input_size).requires_grad_()
158
159     # Zero out gradient, else it will accumulate between epochs
160     optim.zero_grad()
161
162     # Initialises hidden state (initialise after model.train() if LSTM is to be
          stateful)
163     #model.init_states()
```

```
164
165       # Perfoms forward pass
166       yh = model(x)
167
168       # Determines loss
169       loss = criterion(yh, y)
170
171       # Performs backward pass
172       loss.backward()
173
174       # Updates parameters
175       optim.step()
176
177     # Saves loss value in histogram
178     # hist[i] = l.item()
179
180     # Tests the model and reports appropriate metrics
181     # Sets model to test mode
182     model.eval()
183
184     # Initialises helper variables for determining accuracy
185     total   = 0
186     correct = 0
187
188     # Starts inference w/ test set
189     with torch.no_grad():
190       for x, y in test:
191         # Changes shape of x
192         x = x.view(batch_size, 1, input_size)
193
194         # Performs forward pass
195         out = model(x)
196
197         # Determines predicted value
198         yh = torch.argmax(out, 1)
199
200         # Adds to total no. of samples (in case data is batched)
201         total += y.size(0)
202
203         # Determines no. of correct samples
204         correct += (yh == y).sum()
205
206     # Outputs accuracy for given epoch
207     accuracy = 100 * correct / total
208     print('Epoch: {}. Loss: {}. Accuracy: {}'.format(epoch, loss.item(), accuracy
          ))
209
210 # Saves model weights for use in FPGA
211 for param_tensor in model.state_dict():
```

```
212     export_weights(model.state_dict()[param_tensor], param_tensor, "MNIST/lstm"
            , "train", decimal_bits, 'split')
213 print("Weights exported to .bin files with a decimal precision of " + str(
        decimal_bits) + " bits")
214
215 # Saves model weights for use in PyTorch
216 torch.save({'state_dict': model.state_dict()}, '../../model/MNIST/lstm/train/
        checkpoint.pth.tar')
217
218 # Prints confirmation message before exiting
219 print("Training completed!")
```

Listing A.1: Network training in PyTorch. Procedure to dump weights into binary files is included

Appendix B

Resource Utilisation Results

Vivado HLS Synthesis Resource Utilisation Results

S_{batch}	S_{block}	Latency	BRAM_18K		DSP48E		FF		LUT	
125	16	392411620	150	(7.28%)	83	(2.96%)	14712	(2.42%)	77918	(25.66%)
125	32	251812808	202	(9.81%)	147	(5.25%)	18226	(3.00%)	88296	(29.08%)
125	64	233490958	222	(10.78%)	275	(9.82%)	23167	(3.82%)	104497	(34.42%)
125	128	198155971	336	(16.31%)	531	(18.96%)	41408	(6.82%)	135008	(44.47%)
250	16	358786911	150	(7.28%)	83	(2.96%)	14884	(2.45%)	77984	(25.69%)
250	32	207240159	202	(9.81%)	147	(5.25%)	18436	(3.04%)	88349	(29.10%)
250	64	185844223	238	(11.55%)	275	(9.82%)	23505	(3.87%)	104550	(34.44%)
250	128	162585382	432	(20.97%)	531	(18.96%)	41423	(6.82%)	135017	(44.47%)
500	16	341994412	151	(7.33%)	83	(2.96%)	15048	(2.48%)	78037	(25.70%)
500	32	189940220	219	(10.63%)	147	(5.25%)	18663	(3.07%)	88405	(29.12%)
500	64	166846764	335	(16.26%)	275	(9.82%)	23860	(3.93%)	104608	(34.46%)
500	128	150930943	625	(30.34%)	531	(18.96%)	42638	(7.02%)	135145	(44.51%)
1000	16	333637903	167	(8.11%)	83	(2.96%)	15220	(2.51%)	78102	(25.73%)
1000	32	181365991	315	(15.29%)	147	(5.25%)	18898	(3.11%)	88473	(29.14%)
1000	64	157495775	527	(25.58%)	275	(9.82%)	24223	(3.99%)	104678	(34.48%)
1000	128	145395464	977	(47.43%)	531	(18.96%)	43257	(7.12%)	135244	(44.55%)
2000	16	329539144	273	(13.25%)	83	(2.96%)	15392	(2.53%)	78167	(25.75%)
2000	32	177230372	517	(25.10%)	147	(5.25%)	19133	(3.15%)	88541	(29.16%)
2000	64	153115776	889	(43.16%)	275	(9.82%)	24586	(4.05%)	104773	(34.51%)
2000	128	143211220	1723	(83.64%)	531	(18.96%)	43876	(7.23%)	135318	(44.57%)

Table B.1: Latency (in clock cycles) and resource utilisation results for different S_{batch}, S_{block} value pairs ($f = 114.29$ MHz for all pairs) for the simulated accelerator

Board Implementation Resource Utilisation Results

S_{batch}	S_{block}	Latency	BRAM_18K		DSP48E		FF		LUT	
125	64	392371215	111	(5.39%)	275	(9.82%)	11989	(1.97%)	22947	(7.56%)
250	64	325191268	119	(5.78%)	275	(9.82%)	12059	(1.99%)	23275	(7.67%)
500	16	560495301	76	(3.69%)	83	(2.96%)	9654	(1.59%)	16197	(5.33%)
500	32	341070070	110	(5.34%)	147	(5.25%)	10044	(1.65%)	19305	(6.36%)
500	64	292471607	168	(8.16%)	275	(9.82%)	12100	(1.99%)	23834	(7.85%)
500	128	281197055	312	(15.15%)	531	(18.96%)	23857	(3.93%)	31462	(10.36%)
1000	64	276111224	264	(12.82%)	275	(9.82%)	12145	(2.00%)	23664	(7.79%)
2000	64	267931377	445	(21.60%)	275	(9.82%)	12309	(2.03%)	24239	(7.98%)

Table B.2: Latency (in clock cycles) and resource utilisation results for different S_{batch}, S_{block} value pairs ($f = 100\,\text{MHz}$ for all pairs) for the board implementation

Appendix C

C/RTL Cosimulation Waveforms

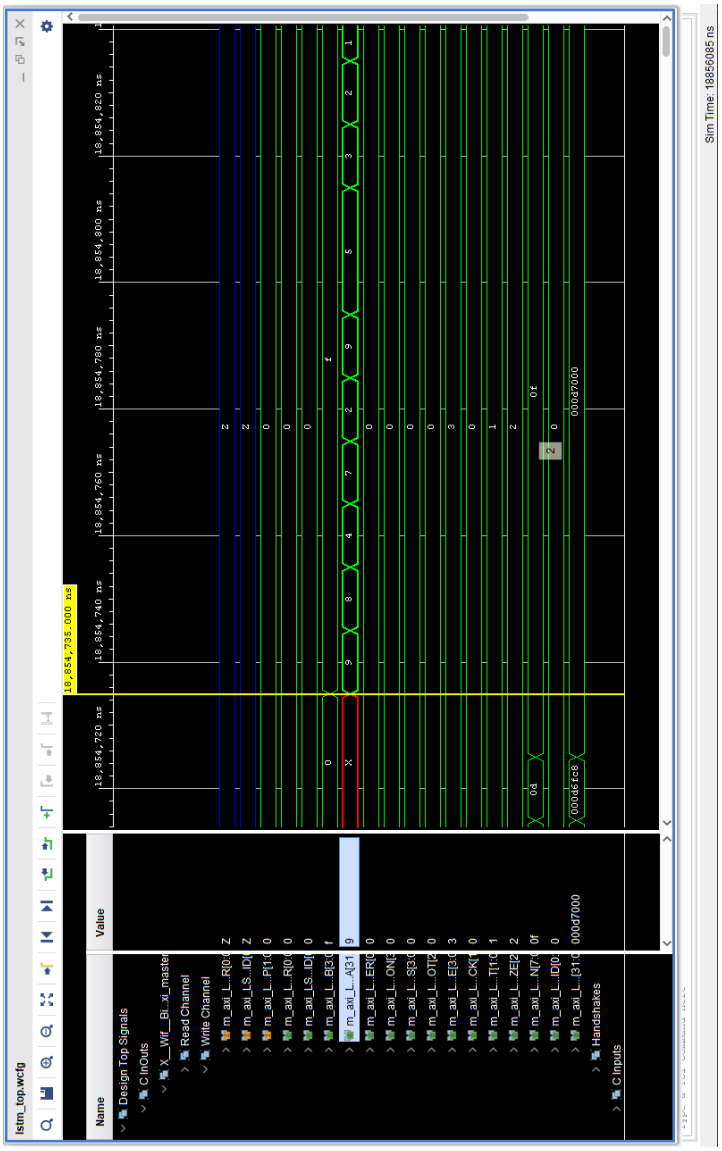


Figure C.1: C/RTL Cosimulation output waveforms from Vivado 2019.1

Bibliography

[1] A. C. Müller, S. Guido *et al.*, *Introduction to machine learning with Python: a guide for data scientists*. " O'Reilly Media, Inc.", 2016.

[2] A. Géron, *Hands-on machine learning with Scikit-Learn and TensorFlow: concepts, tools, and techniques to build intelligent systems*. " O'Reilly Media, Inc.", 2017.

[3] B. Cheng and D. M. Titterington, "Neural networks: A review from a statistical perspective," *Statistical science*, pp. 2–30, 1994.

[4] Q. Le, N. Jaitly, and G. Hinton, "A simple way to initialize recurrent networks of rectified linear units," 04 2015.

[5] C. Olah. Understanding LSTM networks. (27/08/2015, accessed 02/10/2019). [Online]. Available: https://colah.github.io/posts/2015-08-Understanding-LSTMs/

[6] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Comput.*, vol. 9, no. 8, p. 1735–1780, Nov. 1997. [Online]. Available: https://doi.org/10.1162/neco.1997.9.8.1735

[7] F. A. Gers, J. Schmidhuber, and F. Cummins, "Learning to forget: continual prediction with lstm," in *1999 Ninth International Conference on Artificial Neural Networks ICANN 99. (Conf. Publ. No. 470)*, vol. 2, Sep. 1999, pp. 850–855 vol.2.

[8] K. Greff, R. K. Srivastava, J. Koutnik, B. R. Steunebrink, and J. Schmidhuber, "Lstm: A search space odyssey," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 28, no. 10, p. 2222–2232, Oct 2017. [Online]. Available: http://dx.doi.org/10.1109/TNNLS.2016.2582924

[9] Xilinx Inc. Vivado Design Suite User Guide: High-Level Synthesis. (12/07/2019, accessed 25/10/2019). [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2019_1/ug902-vivado-high-level-synthesis.pdf

[10] ——. Introduction to FPGA Design with Vivado High-Level Synthesis. (25/10/2015, accessed 22/01/2019). [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/ug998-vivado-intro-fpga-design-hls.pdf

[11] ——. Vivado Design Suite: AXI Reference Guide. (15/07/2017, accessed 17/01/2020). [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/axi_ref_guide/latest/ug1037-vivado-axi-reference-guide.pdf

[12] A. Graves, M. Liwicki, S. Fernández, R. Bertolami, H. Bunke, and J. Schmidhuber, "A novel connectionist system for unconstrained handwriting recognition," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 31, no. 5, pp. 855–868, May 2009.

[13] S. Fernández, A. Graves, and J. Schmidhuber, "Phoneme recognition in timit with blstm-ctc," *arXiv preprint arXiv:0804.3269*, 2008.

[14] K. Choi, G. Fazekas, and M. Sandler, "Text-based lstm networks for automatic music composition," *arXiv preprint arXiv:1604.05358*, 2016.

[15] V. Flunkert, D. Salinas, and J. Gasthaus, "Deepar: Probabilistic forecasting with autoregressive recurrent networks," *ArXiv*, vol. abs/1704.04110, 2017.

[16] O. Alsharif, T. Ouyang, F. Beaufays, S. Zhai, T. Breuel, and J. Schalkwyk, "Long short term memory neural network for keyboard gesture decoding," in *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2015, pp. 2076–2080.

[17] J. C. Ferreira and J. 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.

[18] S. Wang, Z. Li, C. Ding, B. Yuan, Y. Wang, Q. Qiu, and Y. Liang, "C-lstm: Enabling efficient lstm using structured compression techniques on fpgas," March 2018.

[19] J. Garofolo, L. Lamel, W. Fisher, J. Fiscus, D. Pallett, N. Dahlgren, and V. Zue, "Timit acoustic-phonetic continuous speech corpus," *Linguistic Data Consortium*, November 1992.

[20] S. Cao, C. Zhang, Z. Yao, W. Xiao, L. Nie, D. Zhan, Y. Liu, M. Wu, and L. 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*, ser. FPGA '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 63–72. [Online]. Available: https://doi.org/10.1145/3289602.3293898

[21] M. Marcus, B. Santorini, and M. A. Marcinkiewicz, "Building a large annotated corpus of english: The penn treebank," 1993.

[22] M. Wang, Z. Wang, J. Lu, J. Lin, and Z. Wang, "E-lstm: An efficient hardware architecture for long short-term memory," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 9, no. 2, pp. 280–291, June 2019.

[23] Z. Wang, J. Lin, and Z. Wang, "Hardware-oriented compression of long short-term memory for efficient inference," *IEEE Signal Processing Letters*, vol. 25, no. 7, pp. 984–988, July 2018.

[24] V. Rybalkin, A. Pappalardo, M. M. Ghaffar, G. Gambardella, N. Wehn, and M. Blott, "Finn-l: Library extensions and design trade-off analysis for variable precision lstm networks on fpgas," in *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2018, pp. 89–897.

[25] V. Rybalkin, N. Wehn, M. R. Yousefi, and D. Stricker, "Hardware architecture of bidirectional long short-term memory neural network for optical character recognition," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*, March 2017, pp. 1390–1395.

[26] Z. Que, T. Nugent, S. Liu, L. Tian, X. Niu, Y. Zhu, and W. Luk, "Efficient weight reuse for large lstms," in *2019 IEEE 30th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, vol. 2160-052X, July 2019, pp. 17–24.

[27] D. Kundu, R. Gupta, and A. Manglick, "A convenient way of generating normal random variables using generalized exponential distribution," *Journal of Modern Applied Statistical Methods*, vol. 5, 01 2005.

[28] Minsu Kang, "Fpga implementation of gaussian-distributed pseudo-random number generator," in *6th International Conference on Digital Content, Multimedia Technology and its Applications*, Aug 2010, pp. 11–13.

[29] C. Condo and W. J. Gross, "Pseudo-random gaussian distribution through optimised lfsr permutations," *Electronics Letters*, vol. 51, no. 25, pp. 2098–2100, 2015.

[30] H. Amin, K. M. Curtis, and B. R. Hayes-Gill, "Piecewise linear approximation applied to nonlinear function of a neural network," *IEE Proceedings - Circuits, Devices and Systems*, vol. 144, no. 6, pp. 313–317, Dec 1997.

[31] A. Tisan, S. Oniga, D. Mic, and B. Attila, "Digital implementation of the sigmoid function for fpga circuits," *ACTA TECHNICA NAPOCENSIS Electronics and Telecommunications*, vol. 50, 01 2009.

[32] Y. LeCun, C. Cortes, and C. J. Burges, "The mnist database of handwritten digits, 1998," vol. 10, p. 34, 1998. [Online]. Available: http://yann.lecun.com/exdb/mnist