Graduation Project Report

A Scalable Hardware Architecture for Efficient Sequential Learning at the Edge

Yicheng Zhang, 1720384
y.zhang1@student.tue.nl
Embedded Systems

July 16, 2024

Abstract

Edge devices can increasingly run AI models pre-trained and optimized on large GPU systems. However, these models often require additional fine-tuning to adapt to the nuances of real-life data from the train set. This process, known as edge learning, is vital for personalization and essential for time-series analysis in applications such as speech recognition, gesture recognition, and health monitoring. However, executing sequential learning on edge devices faces the significant challenge of edge systems' limited memory and computing abilities. This work addresses this issue and presents a comprehensive approach to optimize edge sequential learning for resource-constrained systems. We introduce a novel hardware architecture tailored for the Forward Propagation Through Time (FPTT) algorithm that enables memory-efficient sequence processing by partition. To demonstrate our approach, we implement the full computational flow of the FPTT-based training process of a Long Short-Term Memory (LSTM) neural network. Moreover, we propose and optimize a scalable and efficient edge hardware architecture composed of a compressed (BF16 data types) systolic array and multiple RISC-V cores, leveraging an FPGA-based HW/SW co-design flow within Chipyard. Our results show that for small-batch size sequential MNIST training, hardware-accelerated FPTT in BF16 can bring up 22 times saving in memory footprint compared with traditional BPTT training with FP32. Moreover, the scalable architecture can accelerate the training process from 4 to 7 times compared to the basic case but at the cost of up to 4.5 times more resource utilization.

Contents

Acronyms

ANN Artificial Neural Network. 21

ASIP Application-Specific Instruction Set Processor. 21

BPTT Back Propagation Through Time. 1, 7, 11, 16, 32

FPTT Forward Propagation Through Time. 1–3, 5–7, 11, 15, 16, 18–23, 27, 35, 37, 38, 41

MMM Matrix Matrix Multiplication. 20–23, 35, 37

RNN Recurrent Neural Network. 1, 3–5, 11, II, V

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

Egde learning is becoming an increasing need. Nowadays, Edge AI is pervasive as AI models are trained and then deployed on edge devices to solve tasks like object detection and speech command. However, data in real life can be distinct from the train set. For example, unique accents may be a problem for speech assistant AI models trained on general pronunciations. In such a way, AI models are expected to be fine-tuned or trained further to capture the local features precisely and have better performance.

Admittedly, learning on GPU clusters is still feasible to handle the need for post-deployment training. However, training on the edge can have advantages in terms of privacy, latency, and energy compared with transferring user data back to the power-hungry cloud GPU cluster. Therefore, edge learning is necessary to enable AI models to adapt to personalized circumstances in a safe, prompt and efficient way.

1.2 Problems Statement

Sequential Learning of the Recurrent Neural Network (RNN) on the edge is still a challenge for resource-constraint edge devices, as the traditional algorithm, Back Propagation Through Time (BPTT), requires a considerable memory footprint to store all network states generated by the input sequence for backtracking in the training process. Although many bio-inspired temporally local algorithms like E-Prop [1] are applied to edge learning, they have not yet achieved comparable accuracy to BPTT.

Recently, a novel method, Forward Propagation Through Time (FPTT) [2], has been proposed. It evolves from BPTT and adopts the idea of sequence partition, so it is memory-efficient with competitive performance in accuracy, which makes it a promising method for edge learning. However, since it requires precision-sensitive RNNs with the gate structure and is computationally intensive as a spatially non-local algorithm, no edge hardware architecture has been constructed to accelerate FPTT efficiently.

1.3 Reseach Questions

In this work, we focus on two incremental research questions about how to accelerate Forward Propagation Through Time for sequential learning at the edge.

RQ 1 What computations construct the training process based on FPTT algorithm?

- On which RNN FPTT make sense?
- What computations establish forward pass, backward pass, and parameter update of the RNN with FPTT?
- What is the bottleneck to be accelerated in computations?

RQ 2 How to efficiently accelerate computations of FPTT by designing an edge hardware architecture?

- Which type of hardware architecture is suitable for FPTT acceleration, balancing cost, efficiency and flexibility?
- Is it possible to further optimize the hardware architecture for better efficiency considering the features of RNN and FPTT?
- Is it possible to make the hardware architecture scalable, and demonstrate the trade-off between performance and cost when accelerating FPTT?

1.4 Report structure

The result of the report is organized as follows.

- Chapter 2 provides the background and preliminary concepts for later chapters.
- Chapter 3 presents the related works in the context of learning at the edge.
- Chapter 4 investigates the RQ1 and presents the design and analysis of the computation of the training process based on FPTT.
- Chapter 5 inquires into the RQ2, and presents the design and optimization of hardware acceleration of FPTT.
- Chapter 6 concludes the report with contributions, limitations and future works.

Chapter 2

Preliminaries

This section first presents the basic knowledge of recurrent neural networks and the concepts of online learning which are preliminaries to Forward Propagation Through Time (FPTT) algorithm. Secondly, we explain the hardware concepts and Chipyard framework, which are also involved in later chapters.

2.1 Sequential Learning and RNNs

`Sequential Learning` [3] represents the requirement for machine learning models to capture and output the feature of the data sequence. Similarly, `Sequence Modeling` [4] is to use machine learning models to process or generate sequences of data. A sequence consists of multiple time steps, and at each of them, there is a data point being the model input. These data points are not independent and identically distributed (i.i.d.). Instead, they arrive one by one in a specific order, and together carry some dependency and constitute features.

As a pervasive need, sequential learning or sequence modeling exists in considerable real-world scenarios. Typical applications include `Time Series Analysis` like stock market prediction and weather forecasting, and `Natural Language Processing` like machine translation, speech recognition, sentiment classification, language modeling, and text generation for image/video captioning.

2.1.1 Vanilla RNN

Recurrent Neural Networks (RNNs) are the natural fit for sequential learning tasks due to their loop-based structure resembling memory, which can capture the temporal dependency. The loop enables the output of the last time step to become part of the current input, which allows for the persistence of old information in the models. The left part of figure 2.1 presents a vanilla RNN cell along with an output classifier, while their mathematical representations are in Exp.2.1 and Exp.2.2. Note that the optional biases are ignored here for simplicity.

$$h_{(t)} = \sigma_h(x_{(t)} \cdot W^{ih} + h_{(t-1)} \cdot W^{hh}) \tag{2.1}$$

$$\hat{y}_{(t)} = \sigma_o(h_{(t)} \cdot W^{ho}) \tag{2.2}$$

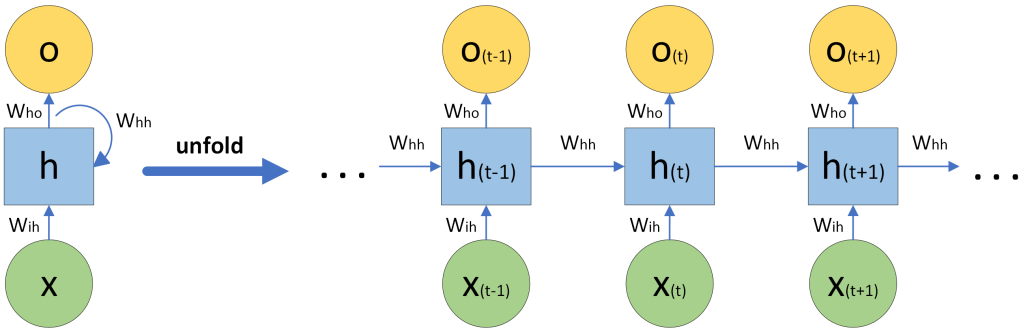

Figure 2.1: Topology of the vanilla RNN and its unfolding

To be specific the input data point at a time step t, i.e. $x_{(t)}$, is multiplied by its weight W^{ih}. Also, the hidden state of the previous time step, $h_{(t-1)}$, is multiplied by W^{hh}. Then, non-linear activation σ_h is performed on the sum of two multiplications, which leads to the hidden state at the current time step $h_{(t)}$. Afterward, the

new hidden state is multiplied by W^{ho}, and with another activation σ_o the final output $\hat{y}_{(t)}$ is obtained. This rule can be scaled to layers of multiple cells.

Types of Tasks: The RNN structure can also be unrolled across the time dimension. As shown in the right part of figure 2.1, the weight is shared within the input, the output, or the hidden state sequence. However, according to the property of tasks, not all $x_{(t)}$ has a meaningful input data p, point, and not all $\hat{y}_{(t)}$ will be collected and used. Specifically, there are four types of tasks [5] in sequence modeling as shown in figure 2.2. The examples for each are given in table 2.1.

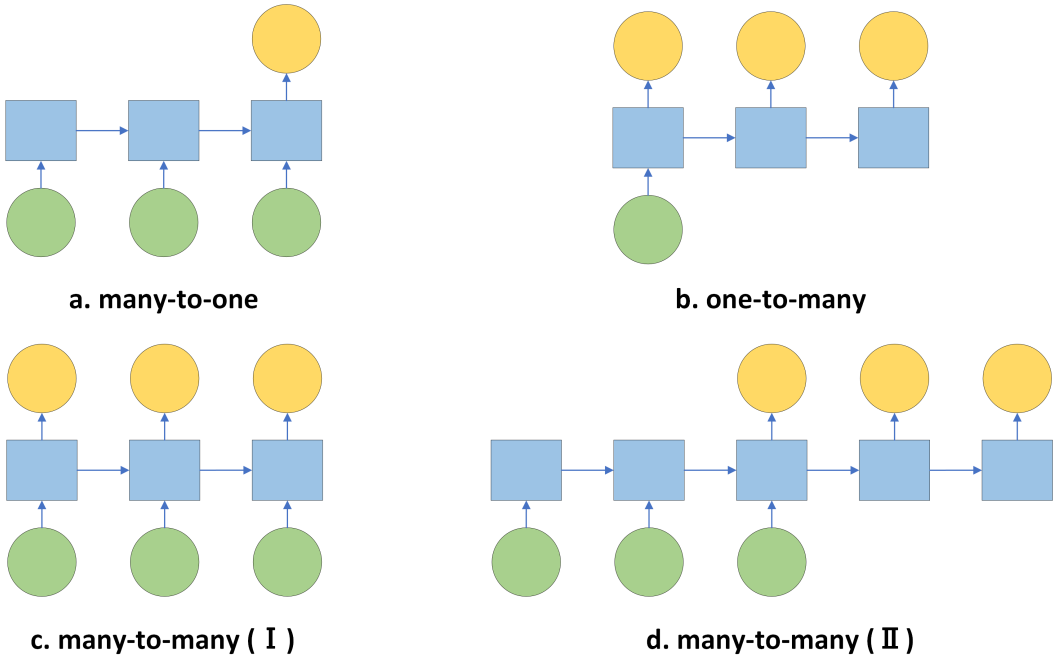

Figure 2.2: Four types of tasks in sequence modeling

Table 2.1: Examples of four sequence modeling tasks

Types	examples
many-to-one [1]	speech command classification, in-pixel image classification
one-to-many	image captioning
many-to-many (I) [2]	language modeling
many-to-many (II) [2]	machine translation

[1] also known as terminal prediction
[2] also known as sequence-to-sequence

Training: `Schocastic Gradient Descent` (SGD) is a typical method to train the RNN and `Backpropagation Through Time` (BPTT) determines the gradient of RNN. Exp.2.3 takes the many-to-many (I) task as an example, in which every output $\hat{y}_{(t)}$ is collected and has a corresponding label $y_{(t)}$, to update the weight W^{hh} in the direction to decrease the difference between $\hat{y}_{(t)}$ and $y_{(t)}$, i.e., loss function L.

$$W^{hh} = W^{hh} - \eta \cdot \frac{\partial L}{\partial W^{hh}} \tag{2.3}$$

The loss function is the summation of the loss at all time steps, so it can be written as Exp.2.4.

$$\frac{\partial L}{\partial W^{hh}} = \sum_{t=1}^{T} \frac{\partial L(y_{(t)}, \hat{y}_{(t)})}{\partial W^{hh}} \tag{2.4}$$

The gradient of Loss at time step t w.r.t. W^{hh} can be expanded by the chain rule. The hidden state at time step t, namely $h_{(t)}$ depends on all previous hidden states.

$$\frac{\partial L(y_{(t)}, \hat{y}_{(t)})}{\partial W^{hh}} = \frac{\partial L(y_{(t)}, \hat{y}_{(t)})}{\partial \hat{y}_{(t)}} \cdot \frac{\partial \hat{y}_{(t)}}{\partial h_{(t)}} \cdot (\sum_{k=1}^{t} \frac{\partial h_{(t)}}{\partial h_{(k)}} \cdot \frac{\partial h_{(k)}}{\partial W^{hh}}) \tag{2.5}$$

To be specific, the gradient of $h_{(t)}$ w.r.t. any hidden state before t can be obtained by the product of partial gradients of two adjacent hidden states.

$$\frac{\partial h_{(t)}}{\partial h_{(k)}} = \prod_{i=k+1}^{t} \frac{\partial h_{(i)}}{\partial h_{(i-1)}} \tag{2.6}$$

Two problems arise from this method of training. First, by Exp.2.6, the product term can have enormous partial gradients for long input sequences. Then unless each partial gradient $\frac{\partial h^{(i)}}{\partial h^{(i-1)}}$ stays close to identity, the product value is prone to vanish or explode, and consequently the value of Exp.2.4. In such a way, the information from the early time steps becomes either trivial or dominant. This is the gradient vanishing or exploding that can lead to poor RNN trainability.

Secondly, by Exp.2.3 and Exp.2.4, the weight is updated in one go after the entire sequence of length T is forwarded into the network. For long input sequences, a large memory footprint would be made to store network states of all time steps.

In addition, `Truncated BPTT` (TBPTT) [6], as a special case of BPTT, is a straightforward way to avoid gradient exploding/vanishing by backpropagating within a fixed number of time steps, which means it is not able to learn the dependency beyond the truncation window.

2.1.2 LSTM

Long Short-Term Memory (LSTM) [7] uses the structure of gates to control the flow of information so that it may keep the information of faraway time steps. Thus, it is employed to capture long-term dependencies and is considered a solution to the problem of vanishing gradients.

Exp.2.7, Exp.2.8 and Exp.2.9 respectively defines the input gate $i_{(t)}$, forget gate $f_{(t)}$ and output gate $o_{(t)}$.

$$i_{(t)} = sigmoid(W^{ix} \cdot x_{(t)} + W^{ih} \cdot h_{(t-1)} + b_i) \tag{2.7}$$

$$f_{(t)} = sigmoid(W^{fx} \cdot x_{(t)} + W^{fh} \cdot h_{(t-1)} + b_f) \tag{2.8}$$

$$o_{(t)} = sigmoid(W^{ox} \cdot x_{(t)} + W^{oh} \cdot h_{(t-1)} + b_o) \tag{2.9}$$

Exp.2.10 defines the candidate cell state $g_{(t)}$.

$$g_{(t)} = tanh(W^{gx} \cdot x_{(t)} + W^{gh} \cdot h_{(t-1)} + b_g) \tag{2.10}$$

Exp.2.11 defines the cell state $s_{(t)}$. The input gate regulates the candidate cell state, while the forget gate regulates the cell state of the previous time step. Then the summation of two multiplications is the cell state.

$$s_{(t)} = g_{(t)} * i_{(t)} + s_{(t-1)} * f_{(t)} \tag{2.11}$$

Last, the output gate controls the flow of the cell state after the activation, which results in the hidden state $h_{(t)}$.

$$h_{(t)} = tanh(s_{(t)}) * o_{(t)} \tag{2.12}$$

2.2 Theories of Learning

Next, we introduce online learning and concepts related to it, since Kag and Saligrama [2] state that online learning inspires Forward Propagation Through Time algorithm.

2.2.1 Learning Paradigms

`Batch Learning` (`Offline Learning`) means training the model with the fixed data set. The entire data set is required to be ready for training. On the other hand, `Online Learning` (`Incremental Learning`) [8] corresponds to the situation that the data point to be learned becomes available one by one with time. The model is supposed to be modified quickly whenever new data becomes available.

`Stream Learning` and `Out-of-Core Learning` are conceptions associated with online learning. Nevertheless, the former emphasizes the condition that the input data streams continuously, while the latter is favored when the large data set cannot fit into the main memory so that few data is fetched and learned every time. In other words, online learning may be applied for different purposes: data to be learned is arriving sequentially; the memory available on the hardware cannot fit the expected batch size. Last but not least, online learning or setting the batch size to 1 may also be adopted for better convergence and accuracy, though without the two mentioned constraints.

`Continual Learning` is a close but different concept since it aims to enable the model to evolve dynamically to learn new data distributions while remembering the learned distributions. Thus, its core is to solve the

catastrophic forgetting problem. On the other hand, both `Offline Learning` and `Online Learning` assume a single distribution of data to be learned.

Delange et al. [9] (Fig.5) explains the difference between `Continual Learning`, `Online Learning`, and other related ways of learning in detail.

2.2.2 Online Learning Algorithms

SGD with backpropagation is the dominant method for both offline learning and online learning in most network architectures. With batch, there are some variants like mini-batch or batch gradient descent depending on the batch size and the size of the data set.

`Online Gradient Descent` (OGD) is the specialized SGD in the context of online learning, in which the time step is also applied to network parameters. Unlike 2.3 that uses the sum $L = \sum_{t=1}^{T} L(y_{(t)}, \hat{y}_{(t)})$ in one go after the complete forward pass, instantaneous Loss $L_{(t)} = L(y_{(t)}, \hat{y}_{(t)})$ is used for gradient descent once available at every time step, as shown in Exp.2.13. In other words, online learning interleaves forward pass and backward pass, leading to T rather than 1 update on network parameters during the training of sequence of length T.

$$W^{hh}_{(t+1)} = W^{hh}_{(t)} - \eta \cdot \frac{\partial L_{(t)}}{\partial W^{hh}_{(t)}} \tag{2.13}$$

`Follow the Regularized Leader` (FTRL) [10] is another classical method of online learning. It attempts to find a direction to update the network parameter by cumulative loss of all time steps so far, unlike OGD that only uses the loss at the current time step. Moreover, a regularization term $R(w)$ is introduced for stabilization.

$$w_{(t+1)} = argmin_{w \epsilon S} \sum_{i=1}^{t} L_i(w) + R(w) \tag{2.14}$$

2.3 Forward Propagation Through Time

Then we explain Forward Propagation Through Time (FPTT) algorithm based on the background mentioned previously.

2.3.1 Algorithm

We first introduce BPTT in Algorithm 1from which FPTT evolves. Then FPTT is elaborated incrementally in Algorithm 2. In both cases, the sequence-to-sequence problem is considered, which means there is an output and label at every time step.

In each training step of BPTT, the forward pass is performed continuously on the entire input sequence of length T. In such a way, network states h_t at every time steps t are stored. Afterward, we take the gradient of the summed loss w.r.t. the current weight to update the weight as gradient descent works. Also, if truncation is not specified in the backward pass, backpropagation from any time step t will trace back to all network states before it, until the first state, as the term $\sum_{k=1}^{t} \frac{\partial h_{(t)}}{\partial h^{(k)}}$ in Exp.2.5.

Algorithm 1 Training RNN with BPTT

Input: Training data $B = \{x^i, y^i\}_{i=1}^{N}$, Timesteps T
Input: Learning rate η, Hyper-parameter α, # Epoch E
Initialize: W_1 randomly in the domain $\mathcal{W}$
for $e = 1$ **to** E **do**
    randomly shuffle B
    for $i = 1$ **to** N **do**
        Set: $(x, y) = (x^i, y^i)$ and $h_0 = 0$
        for $t = 1$ **to** T **do**
            Update: $h_t = f(W, x_t, h_{t-1})$
        end for
        Loss: $l(W) = \sum_{t=1}^{T} l(y_t, v^T h_t)$
        Set: $W_{i+1} = W_i - \eta \nabla_W l(W)|_{W=W_i}$
    end for
    Reset: $W_1 = W_{N+1}$
end for
Return: W_{N+1}

On the other hand, FPTT interleaves the forward pass and backward pass since it follows Online Learning.

Algorithm 2 Training RNN with FPTT

Input: Training data $B = \{x^i, y^i\}_{i=1}^N$, Timesteps T
Input: Learning rate η, Hyper-parameter α, # Epoch E
Initialize: W_1 randomly in the domain $\mathcal{W}$
Initialize: $\overline{W}_1 = W_1$
for $e = 1$ **to** E **do**
  randomly shuffle B
  for $i = 1$ **to** N **do**
    Set: $(x, y) = (x^i, y^i)$ and $h_0 = 0$
    for $t = 1$ **to** T **do**
      Update: $h_t = f(W, x_t, h_{t-1})$
      $l_t(W) = l_t(y_t, v^T h_t)$
      $l(W) = l_t(W) + \frac{\alpha}{2}\|W - \overline{W}_t - \frac{1}{2\alpha}\nabla_{l_{t-1}}(W_t)\|^2$
      $W_{t+1} = W_t - \eta \nabla_W l(W)|_{W=W_t}$
      $\overline{W}_{t+1} = \frac{1}{2}(\overline{W}_t + W_{t+1}) - \frac{1}{2\alpha}\nabla_{l_t}(W_{t+1})$
    end for
    Reset: $W_1 = W_T$ and $\overline{W}_1 = \overline{W}_T$
  end for
end for
Return: W_T

To begin with, another group of network parameters is defined as the running average, i.e., $\overline{W}$ that corresponds to W. At any time step t of a training step i, the model performs the forward pass for one time step.

The generated output and the label construct the instantaneous loss function $l_t(W)$ as BPTT does. Moreover, the total loss $l(W)$ also includes a regularization term $\frac{\alpha}{2}\|W - \overline{W}_t - \frac{1}{2\alpha}\nabla_{l_{t-1}}(W_t)\|^2$, $R(W)$ for shorthand. It measures the difference between the current weight and the running average of the weight. So, being a part of the loss function, $R(w)$ penalizes the difference and encourages the weight to be close to its running average to stabilize the training process.

Then, the backward pass is also performed for one time step of the generated network states. After updating the weight by the gradient of total loss, the running average $\overline{W}$ is updated by averaging the new weight and the history average. So, the weight value at a later time step is a larger priority in the running average.

This routine would repeat T times for the sequence of length T.

Furthermore, both $R(W)$ and $\overline{W}$ use the same term, namely $\nabla_{l_{t-1}}(W_t)$ or $\nabla_{l_t}(W_{t+1})$ that is claimed to be a small correction term. It requires the gradient of old loss w.r.t. new weight. It is suggested to be kept as a running estimate λ_t in Exp. 2.15. And $\lambda_0 = 0$.

$$\lambda_{t+1} = \lambda_t - \alpha(W_{t+1} - \overline{W}_t) \tag{2.15}$$

Remarks: In general, Forward Propagation Through Time evolves from BPTT and absorbs online learning, `Follow the Regularized Leader` (FTRL) and `Truncated BPTT` (TBPTT), because the forward pass and the backward pass are interleaved, and a regularization term is applied for stabilization, and backward pass uses network states of one time step only.

In terms of memory, the main difference between FPTT with BPTT is that the former keeps two extra parameter groups, running average $\overline{W}$ (optionally bias) and running estimate λ_t(or $\nabla_{l_{t-1}}(W_t)$). They have the same size as network parameters (weights and biases). Therefore, one downside of FPTT is that the number of network parameters has to be tripled.

In addition, the update of $\overline{W}$ and λ_t is done immediately after the weight update, so it seems natural to integrate them into the optimizer in terms of implementation.

2.3.2 Generalization

FPTT theory also includes two useful generalizations, i.e., FPTT-K and auxiliary loss.

The first is FPTT-K. For BPTT in Algorithm 1, the forward pass goes for T time steps, and then the backward pass uses generated T network states. In Algorithm 2, the forward pass proceeds one time step, and the backward pass uses one network state in T iterations.

However, there are granularities in between. For example, we can let the forward pass go for two steps, then do a backward pass on the generated two network states and update the network parameter. In such a way, the network parameters are updated less frequently although more states should be stored.

In general, FPTT-K is proposed, in which K intuitively means the input sequence of length T is partitioned into K parts and each has the length of $\frac{T}{K}$. In algorithm 2, K is default as sequence length T, which means the input sequence is partitioned into T parts, each having one time step, as we do forward pass, backward pass and then parameter update for every time step.

Moreover, Kag and Saligrama [2] mention that K being $\sqrt{T}$ achieves best trainability.

A special case is when the division has a reminder. For example, when K is 6 and T is 100, so we want to partition 100 into 6 parts. To that end, the floor division would be applied. As $100/6 = 16; 100 - 16 * 6 = 4$, there would be 6 parts of length 16 and 1 part of length 4. In other words, 7 parts in total. For simplicity, division with a reminder would be ignored.

Another expansion is auxiliary loss. Previous explanations are based on sequence-to-sequence tasks in which there is a label at each time step. However, in terminal prediction applications like image classifications, one input sequence has only one label. Then, to enable FPTT on terminal predictions, intermediate labels are created, as in Exp. 2.16 and 2.17.

$$l_t = \beta \cdot l_t^{CE} + (1 - \beta) \cdot l_t^{Div} \tag{2.16}$$

$$l_t^{CE} = -\sum_{\overline{y} \in y} \mathbf{1}_{\overline{y}=y} log\hat{P}(\overline{y}); \quad l_t^{Div} = -\sum_{\overline{y} \in y} Q(\overline{y}) log\hat{P}(\overline{y}); \quad \beta = \frac{t}{T} \tag{2.17}$$

$\hat{P}$ is the current estimate of the label distribution or the network output, so the classification loss l_t^{CE} is the cross entropy loss between the network output and the label.

Q is estimated in the last training epoch, then auxiliary loss/oracle loss l_t^{Div} is the cross-entropy loss between the current network output and the output in the last epoch.

With β being the current time step divided by the sequence length, it follows that towards a long input sequence the prediction is first forced to be close to the history prediction to keep stable and then gradually approaches the real label with time steps going.

2.3.3 Cost

Kag and Saligrama [2] provide the comparison (table 2.2) of FPTT(-K) with BPTT/FTRL about three types of cost, assuming that the task is the sequence-to-sequence problem. Note that the constant associated with the gradient, computation is a monotonically increasing function $c(.)$ of the sequence length, i.e. $c(1) < c(K) < c(T)$.

Table 2.2: Computational cost for gradient, parameter update & memory storage overhead

Algorithm	Gradient Updates	Parameter Updates	Memory Storage
BPTT	$\Omega(c(T)T)$	$\Omega(1)$	$\Omega(T)$
FTRL	$\Omega(c(T)T^2)$	$\Omega(T)$	$\Omega(T)$
FPTT	$\Omega(c(1)T)$	$\Omega(T)$	$\Omega(1)$
FPTT-K	$\Omega(c(K)T)$	$\Omega(K)$	$\Omega(T/K)$

As can be observed from the table 2.2, FPTT trades more frequent parameter updates for less memory usage and less gradient calculations in the field of sequence-to-sequence tasks. Also, higher accuracies are reported. Nevertheless, the gradient calculations are not reduced for terminal prediction problems.

2.4 Execution on Hardware

2.4.1 Trade-off in Hardware Acceleration

Figure 5 presents the trade-off between performance and flexibility when designing hardware to accelerate an algorithm.

2.4.2 Arithmetics

The IEEE Standard for Floating-Point Arithmetic (IEEE 754) [11] is a technical standard for floating-point arithmetic established in 1985. `Universal Numbers` (Unums) [12] is a family of number formats and arithmetic for implementing real numbers on a computer which are designed as an alternative to the ubiquitous IEEE 754 floating-point standard.

Langroudi, Carmichael, and Kudithipudi [13] demonstrates that posit numerical format (Type III Unum) has high efficacy for DNN training at f16, 32g-bit precision training, surpassing the equal-bandwidth fixed-point

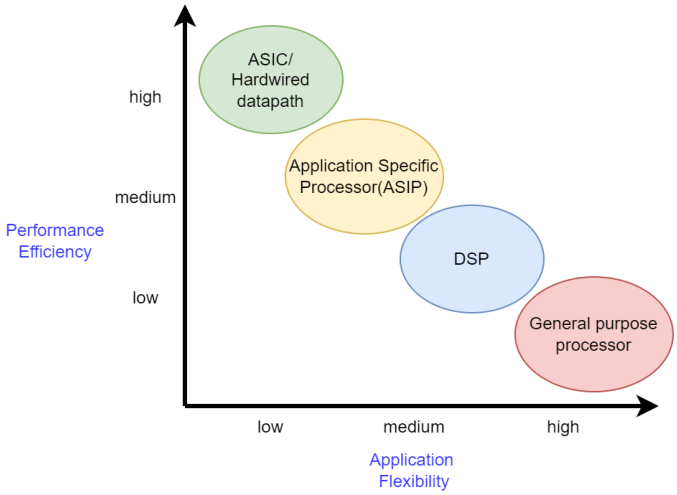

Figure 2.3: Trade-off between performance and flexibility of the hardware (from course 5LID0)

and floating-point counterparts. Notably, Posit-16 has obvious advantages in terms of accuracies on training Float-16 on MNIST and Fashion MNIST dataset (96.54% vs 90.65%; 87.40% vs 81.73%).

`BFLOAT16` (brain floating point 16-bit) is another promising data format for neural network training. Kalamkar et al. [14] presents the first comprehensive empirical study demonstrating the efficacy of the BFLOAT16 half-precision format for Deep Learning training across image classification, speech recognition, language modeling, generative networks and industrial recommendation systems. Their results show that deep learning training using BFLOAT16 tensors achieves the same state-of-the-art results across domains as FP32 tensors in the same number of iterations and with no changes to hyper-parameters.

2.5 Chipyard

Chipyard [15] is a framework for designing and evaluating full-system hardware using agile teams. It comprises a collection of tools and libraries designed to integrate open-source and commercial tools for developing systems-on-chip, covering hardware design, RTL simulation, FPGA prototyping, VLSI flow and software development.

2.5.1 SoC template

For hardware design, a series of Hardware IPs are available for users to construct a system at their will, including some RISC-V-based CPUs and other domain-specific architectures like Gemmini. Especially, hardware IPs are usually called the generator as they are implemented in Chisel [16], a scala-based hardware construction language that generates low-level Verilog designed to map to either FPGAs or a standard ASIC flow for synthesis.

Figure 2.4 presents a template of the Chipyard-based system using available hardware blocks, in which the size of caches is from the default configuration. To begin with, the interconnect or System Bus, based on the `TileLink` standard, assembles all subsystems. Then each `RocketTile` is a CPU subsystem that contains a CPU core, data cache, instruction cache, Page Table Walker (PTW) and the interface to the System Bus. The CPU instance here is Rocket [17] which is a 5-stage in-order scalar RISC-V core generator that implements the RV32G and RV64G ISAs. Alternatives can be Berkeley Out-of-Order (BOOM) Core [18], CVA6 [19] (previously called Ariana), Ibex[1], etc.

Besides, we can attach the RoCC (Rocket-Chip Co-processor) accelerator, i.e. Gemmini, to the `RocketTile`. Generally, suppose the instruction decoder of the CPU detects a custom instruction for an available RoCC accelerator. In that case, that instruction will be sent to the accelerator for execution, during which the CPU will be waiting. Another channel in between transfers the address mapping. Moreover, the RoCC accelerator bypasses L1 caches and accesses the L2 cache directly for high bandwidth via the System Bus. The cache coherency is handled by `TileLink` protocol [2].

The L2 cache subsystem contains multiple banks, it can be connected to an AXI-interfaced DRAM controller through a TileLink to AXI converter. For MMIO (Memory Mapped I/O) peripherals, the System Bus connects to the Control Bus and Periphery Bus. Control Bus connects to blocks that handle the bootloader, various

[1] `https://ibex-core.readthedocs.io/en/latest/index.html`
[2] `https://sifive.cdn.prismic.io/sifive%2Fcab05224-2df1-4af8-adee-8d9cba3378cd_tilelink-spec-1.8.0.pdf`

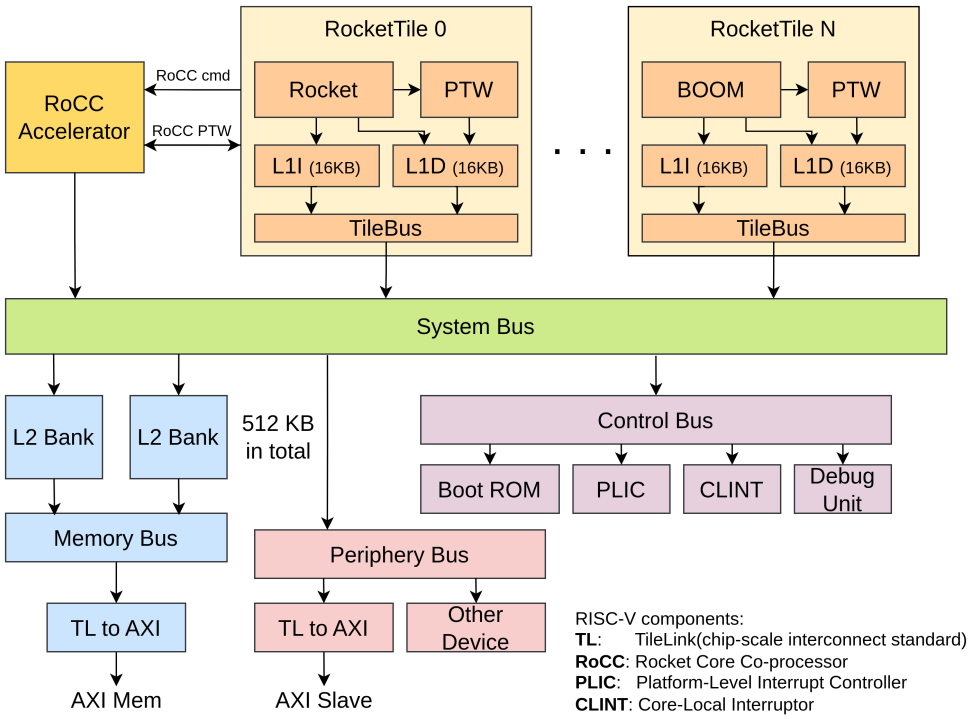

Figure 2.4: An example of chipyard-based SoC

interrupts and external control of the chip. Periphery Bus integrates other devices with either `TileLink` or AXI interface.

2.5.2 Simulation

Simulation is a critical method of functional verification, and three simulation flows are provided within the Chipyard, as shown in table 2.3 and figure 2.5.

Table 2.3: Methods of Simulation in Chipyard

Type	Method	Compile	Run	Waveform Support	Cycle-Accurate
Instruction Simulation	Spike	N/A	fast	N/A	No
RTL Simulation	Verilator/VCS	fast	slow	Easy	Yes
FPGA-accelerated HW/SW Co-Simulation	FireSim	slow	fast	Hard	Yes

`Spike` is the golden reference functional RISC-V ISA C++ software simulator. It can execute the binary file of the application in a speedy way and provide results just like a real RISC-V platform. `Spike` is also highly scalable, as it supports multicore and custom instructions of some RoCC accelerators. However, as an instruction simulator, it does not model microarchitecture and is not cycle-accurate. Specifically, it usually acts as a starting point for RISC-V-targeted software development on a non-RISC-V platform to examine the functional correctness of workloads.

RTL simulation is also well-supported in Chipyard. Scripts are provided for the open-source simulator (`Verilator`) and the commercial simulator (`VCS`, `Xcelium`). Compilation of the design by the RTL simulator is relatively fast, while the cycle-accurate run is slow. Nevertheless, the advantage is that the waveform of all cycles can be kept to help debugging. Thus, RTL simulation is used to run and debug simple workloads.

`FireSim` is a unique technique of FPGA-accelerated HW/SW co-simulation in Chipyard. The design is synthesized as bitstream and deployed on the FPGA board. Unlike normal FPGA prototyping in which real peripheral devices are used, `FireSim` utilizes software models to have deterministic execution results. Execution on the FPGA board can be tens of megahertz, though synthesis is still time-consuming. In addition, waveform support is limited. In general, `FireSim` is expected to profile the performance of large workloads quickly and deterministically.

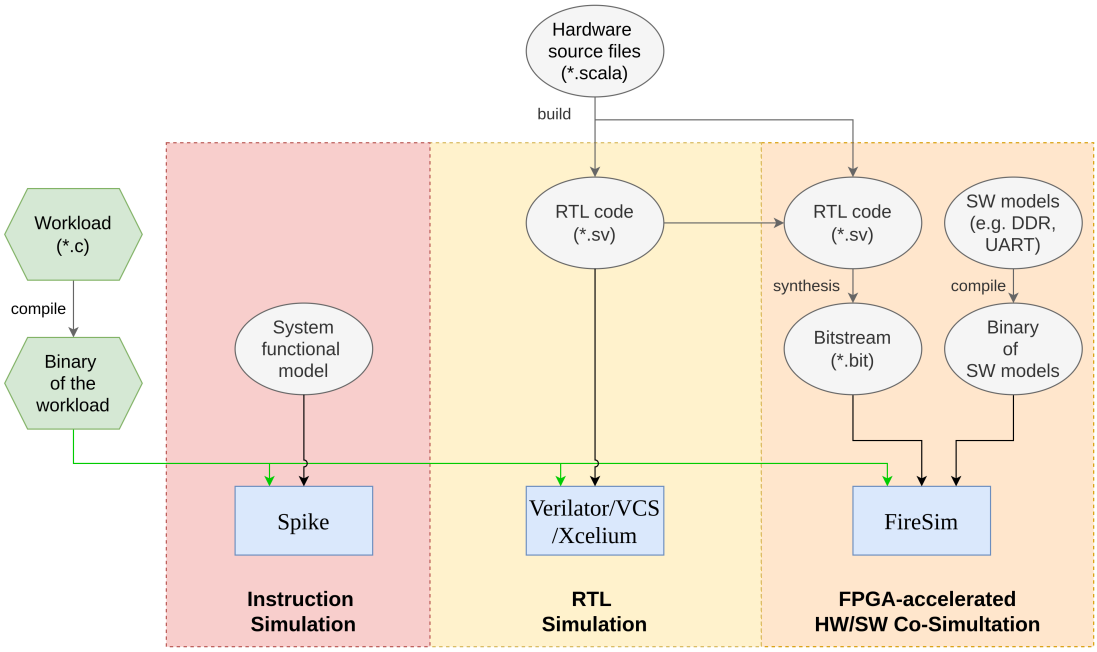

Figure 2.5: Flows of three simulation methods

2.6 Summary

In this chapter, we first introduce the Forward Propagation Through Time (FPTT). It is a training algorithm for RNN, which evolves from Back Propagation Through Time (BPTT) and absorbs Online Learning, `Follow-the-Regularised-Lead` and `Truncated BPTT`. It can be more generalized by FPTT-K, with K being the partition factor and auxiliary loss for the terminal prediction task. It trades more parameter updates for less memory footprint.

We also mention the trade-off in hardware acceleration and promising data types for neural networks like BF16. Last, we explain Chipyard, a framework for agile SoC development, and the simulation infrastructure.

Chapter 3

Related Work

In this section, we introduce related work in RNN training acceleration and edge learning.

3.1 Hardware Acceleration of RNN Training

RNN inference accelerations are carried out in ASICs or FPGA. At the same time, GPU still dominates RNN training of large batches, along with software techniques to exploit GPU for higher efficiency and utilization. For example, weight pruning and low-precision computation are widely applied.

On the other hand, Cho, Lee, and Lee [20] points out that for some situations where the small batch size is preferred, e.g. partial retraining for personalization, GPU performance suffers from low utilization. Therefore, they propose a hybrid architecture FARNN that combines GPU and FPGA. Specifically, GPU-inefficient tasks are off-loaded to dedicated computation units in the FPGA. The evaluation result indicates that FARNN outperforms the P100 GPU platform for RNN training by up to 4.2x with small batch sizes and long input sequences.

Li et al. [21] presents an FPGA implementation framework for RNNLM training acceleration. At the architectural level, they improve the parallelism of the RNN training scheme and reduce the computing resource requirement for computation efficiency enhancement. The hardware implementation primarily targets reducing data communication load. A multi-thread-based computation engine is utilized which can successfully mask the long memory latency and reuse frequently accessed data.

3.2 Edge Learning

We investigate five works in neural network training on edge, where various algorithm innovations are proposed to fit the model and the training process on resource-constrained devices. They are summarized in table 3.1. This is a relatively new domain, and most works accelerate CNN while still inspiring us.

Lin et al. [22] propose an algorithm-system co-design framework to make on-device training possible with only 256KB of memory, i.e. finetuning a pre-trained model. Quantization Aware Scaling to calibrate the gradient scales and stabilize 8-bit quantized training. To reduce the memory footprint, they propose Sparse Update to skip the gradient computation of less important layers and sub-tensors. The integration of the algorithm, Tiny Training Engine, enables the use of less than 1/1000 of the memory of PyTorch and TensorFlow while matching accuracy.

Ren, Anicic, and Runkler [23] also base their work on an MCU. They propose a novel system called TinyOL (TinyML with Online Learning), which enables incremental on-device training of an autoencoder model on streaming data by fine-tuning the last fully connected layer.

Ravaglia et al. [24] introduce a HW/SW platform for end-to-end Continual Learning based on a 10-core FP32-enabled parallel ultra-low-power (PULP) processor. They leverage quantization of the frozen stage of the model and Latent Replays (LRs) to reduce their memory cost with minimal impact on accuracy. In particular, 8-bit compression of the LR memory proves to be almost lossless (-0.26% with 3000LR) compared to the full-precision baseline implementation. On the 22nm prototype of the VEGA platform, the proposed solution performs on average 65× faster than a low-power STM32 L4 microcontroller, being 37× more energy efficient.

`Checkpointing` (or reforwarding) means a subset of activations is stored during the forward pass, and the rest is discarded. The discarded data can be recovered by rerunning the forward propagation from the last available "checkpoint". Kukreja et al. [25] shows that with customized checkpointing, the memory footprint can be kept within 2GB for ResNet training (up to 152 layers, up to the batch size of 8, up to image size 500).

Table 3.1: Related work of training on the edge device

Paper	Network	Task	Hardware
Lin et al. [22]	MobileNetV2-w0.35 ProxylessNAS-w0.3 MCUNet (5FPS)	ImageNet Visual Wake Words	STM32F746 Cortex-M7 320KB-SRAM 1MB-Flash
Ren, Anicic, and Runkler [23]	autoencoder NN	the modes of vibration of a fan	Arduino Nano 33 BLE Sense: Cortex-M4, 256KB SRAM, 1MB flash
Ravaglia et al. [24]	MobileNet-V1	Core50	VEGA: A 10-core RISC-V processor, 64MB SRAM
Kukreja et al. [25]	from ResNet-18 to ResNet-152	image classification	ODROID XU4 board: 4 A15 cores, 4 A7 cores, Mali-T628 MP6 GPU, 2GB LPDDR3 RAM
Yuan et al. [26]	ResNet-32	CIFAR-100	Samsung Galaxy S20 smartphone: Qualcomm Adreno 650 mobile GPU

Yuan et al. [26] propose Memory-Economic Sparse Training (MEST) targeting for accurate and fast execution on edge devices. The proposed MEST framework consists of enhancements by Elastic Mutation (EM) and Soft Memory Bound (&S) that ensure superior accuracy at high sparsity ratios.

3.3 Commercial Edge AI hardware

The Edge AI hardware covers a wide range of budgets for various requirements. Inspired by Burgt [27], we list a few products having an architecture similar to ours in table 3.2, i.e., multiple CPUs, one domain-specific accelerator, 1MB-level of on-chip SRAM and a few megabytes of external memory.

Table 3.2: Edge AI products

Product	processor(s)	SRAM	Flash/DRAM	Price
Sony `Spresense Main Board`	6-core Arm-Cortex-M4F(156MHz)	1.5MB	8MB/-	$65
Arduino `Portenta H7`	Arm-Cortex-M7(480MHz) Arm-Cortex-M4(240MHz) Chrom-ART Graphics Accelerator	1MB	16MB/8MB	$99
Greenwaves `GAP8`	8-core 64-bit RISC-V[1]cluster(175MHz) CNN Accelerator	580KB	64MB/8MB	$60
Greenwaves `GAP9`	9-core 64-bit RISC-V[2]cluster(370MHz) Cooperative AI Accelerator	1.6MB	-/-	$90
SiPEED `MAix Go`	2-core 64-bit RISC-V (400MHz) CNN Accelerator	8MB	16MB/-	$40
NXP i.MX 8ULP SOM	2-core Arm-Cortex-A35 (800MHz) Arm-Cortex-M33 (216MHz) Tensillica Hi-Fi 4 DSP (475MHz) Fusion DSP (200MHz)	896KB	-/-	$114

[1] Extended instructions include SIMD vector, DSP and bit manipulation
[2] Support FP32, FP16 and BF16

3.4 Summary

Deploying neural network training on resource-constrained platforms is challenging, and both hardware and software need to be considered for two main challenges: reasonable latency of heavy computational load on limited computing resources and memory-inefficient training processes on memory-limited platforms. The work mentioned involves five types of effort.

- **Identify the feature of computation and customize hardware for it**. Cho, Lee, and Lee [20]

identify the GPU-inefficient tasks and off-load them to dedicated computing units in FPGA. Li et al. [21] implement the hardware targeting the reduction of data communication load.

- **Fully exploit the parallelism in the computation for acceleration**. Li et al. [21] improves parallelism of the training scheme at the architectural level.
- **Skip the unnecessary computations**. Lin et al. [22] skip the gradient computation of less important layers and sub-tensors and Ren, Anicic, and Runkler [23] only fine-tunes the last fully connected layer.
- **Trade more computations for less memory**. Latent Replay is used by Ravaglia et al. [24] and Reforwarding (Checkpointing) is used by Kukreja et al. [25].
- **Use compact data types to save memory**. Ravaglia et al. [24] uses 8-bit compressed latent replay memory.

Chapter 4

Computation Design

This chapter presents the methodology and results of the computation design of FPTT.

4.1 Methodology

The methodology has two steps. To begin with, we determine the computation of the neural network. Afterward, we construct the workload of the training process by C programming. A working C program can be versatile, as we can perform profiling on it to understand the computation bottleneck. In later phases, it can either run on a general-purpose platform like a CPU or bridge the gap between the algorithm and HLS/RTL implementation.

4.1.1 Neural Network Implementation

In this part, we figure out the network to be used, including its forward pass, backward pass, and parameter updates. Parameter updates include core calculations of FPTT algorithm, i.e., the running average and the running estimate.

Network Structure

We first figure out the network structure. FPTT is expected to solve sequential learning with RNNs, then the RNN should be the core layer inside the network. Yin, Corradi, and Bohté [28] point out that a straightforward application of FPTT to vanilla RNNs fails; they deduce that FPTT particularly benefits from the gating structure inherent in LSTMs and GRUs, which is lacking in vanilla RNNs. Moreover, Kag and Saligrama [2] shows that LSTM has a better performance than GRU. So, we choose LSTM as the backbone.

Then, we simplify the setting of Kag and Saligrama [2] to a lightweight network in table 4.1 for edge computing, by reducing 2 linear layers to 1 and removing the dropout calculations.

Table 4.1: Network structure

Layer Index	Layer Type
1	LSTM
2	Linear [1]

[1] followed by `softmax`

Forward Pass

Then, we derive the forward pass of the aforementioned layer structure. In the previous section 2.1.2, we list the forward pass of LSTM, while we rewrite those expressions here to make it easier to implement by adding the concatenation and more intermediate stages.

We note $x_{(t)}$ as the input to network at time step t, and $xc_{(t)}$ is the concatenation of $x_{(t)}$ and the hidden state of last time step $h^{l1}_{(t-1)}$. As the input and the hidden states are merged, their weight matrices W^i, W^f, W^g, W^o are also concatenated for the candidate cell $g_{(t)}$, input gate $i_{(t)}$, forget gate $f_{(t)}$ and output gate $o_{(t)}$. Note that notation $[\cdot]$ means matrix multiplication, while $[*]$ represents element-wise/vector multiplication(a.k.a. Hadamard product).

$$xc_{(t)} = [x_{(t)}, h_{(t-1)}] \tag{4.1}$$

$$W^{\star} = [W^{x\star}, W^{h\star}](\star = i, f, g, o) \tag{4.2}$$

$$g_{(t)} = tanh(g\_input_{(t)}); \qquad g\_input_{(t)} = \mathbf{W}^{g} \cdot xc_{(t)} + b^{g} \tag{4.3}$$

$$i_{(t)} = sigmoid(i\_input_{(t)}); \qquad i\_input_{(t)} = \mathbf{W}^{i} \cdot xc_{(t)} + b^{i} \tag{4.4}$$

$$f_{(t)} = sigmoid(f\_input_{(t)}); \qquad f\_input_{(t)} = \mathbf{W}^{f} \cdot xc_{(t)} + b^{f} \tag{4.5}$$

$$o_{(t)} = sigmoid(o\_input_{(t)}); \qquad o\_input_{(t)} = \mathbf{W}^{o} \cdot xc_{(t)} + b^{o} \tag{4.6}$$

$$s_{(t)} = g_{(t)} * i_{(t)} + s_{(t-1)} * f_{(t)} \tag{4.7}$$

$$h^{l1}_{(t)} = tanh(s_{(t)}) * o_{(t)} \tag{4.8}$$

$$h^{l2}_{(t)} = \mathbf{W}^{l2} \cdot h^{l1}_{(t)} + b^{l2} \tag{4.9}$$

$$\hat{y}_{(t)} = softmax(h^{l2}_{(t)}) \tag{4.10}$$

Backward Pass

We then derive the backward pass of the aforementioned network, i.e., the process of finding the gradient of losses w.r.t. the network parameter.

Since the loss function in FPTT theory consists of the normal loss and the regularization term $R_{(t)}$, their gradients are derived separately. Normal loss is network-dependent, so we can derive its gradient by BPTT. On the other hand, the regularization term $R_{(t)}$ is a straightforward function of network parameters, which is independent of network topology, so we can figure out the gradient by direct differentiation. Note that Kag and Saligrama [2] implements the FPTT by `PyTorch` that adopts runtime auto-differentiation.

Algorithm 3 organizes the flow to find the gradient of normal loss by backtracking (BPTT) starting from the given time step t, as it returns $NormalGrad$ eventually. 4.11 and 4.12 are the form of shorthand applied. θ is the network parameters, including the weight and the bias, and *state* are the network states at a time step, i.e., h^{l1}.

$$\mathbf{d}\theta = \frac{\partial L_{(t)}}{\partial \theta} \qquad (\theta = \{W^{\star}, b^{\star}\}, \quad \star \in \{i, f, g, o, l2\}) \tag{4.11}$$

$$\mathbf{d}state = \frac{\partial L_{(t)}}{\partial state_{(i)}} \qquad (0 \le i \le t, \quad state \in \{i, f, g, o, s, h^{l1}, h^{l2}\}) \tag{4.12}$$

Secondly, we find the gradient of the regularization term $R_{(t)}$ which is defined as $\frac{\alpha}{2}\|W - \overline{W}_t - \frac{1}{2\alpha}\nabla_{l_{t-1}}(W_t)\|^2$ in algorithm 2. Since we use biases as well, the expression can be generalized as Exp. 4.13.

$$R_{(t)} = \frac{\alpha}{2}\|\theta - \overline{\theta}_{(t)} - \frac{1}{2\alpha}\nabla_{l_{t-1}}(\theta_{(t)})\|^2 \qquad (\theta \in \{W^{\star}, b^{\star}\}, \quad \star \in \{i, f, g, o, l2\}) \tag{4.13}$$

Running mean $\overline{\theta}_t$ and running estimate $\nabla_{l_{t-1}}(\theta_t)$ are considered constants with regard to θ in a alternative optimization method [2], then in Exp.4.14 we derive $R'_{(t)}$, namely the gradient of $R_{(t)}$ w.r.t. θ by direct differentiation.

$$R'_{(t)} = \alpha(\theta - \overline{\theta}_{(t)}) - \frac{1}{2}\nabla_{l_{t-1}}(\theta_{(t)}) \tag{4.14}$$

We can further simplify the expression in Exp.4.15 since λ_t is the shorthand for $\nabla_{l_{t-1}}(\theta_t)$.

$$R'_{(t)} = \alpha(\theta - \overline{\theta}_{(t)}) - \frac{1}{2}\lambda_{(t)} \tag{4.15}$$

To have a cross-verification on our derivation, we also check the implementation published by Kag and Saligrama [2], and the function that realizes $R_{(t)}$ is in listing 4.1. The comments are added by us.

```
def get_regularizer_named_params( named_params, args, _lambda=1.0 ):
    alpha = args.alpha
    rho = args.rho # is set to 0.0
    regularization = torch.zeros( [], device=args.device )
    for name in named_params:
        param, sm, lm, dm = named_params[name]
        # parm: network parameters, weight/bias
        # sm: running average
        # lm: running estimate, lambda
        # dm: not used
        regularization += (rho-1.) * torch.sum( param * lm )
        if args.debias: # is set to false, ignore
            regularization += (1.-rho) * torch.sum( param * dm )
        else:           # branch selected
            regularization += _lambda * 0.5 * alpha * torch.sum( torch.square(param - sm) )
```

Algorithm 3 BPTT of the network

Input: the time step to start backpropagation: t
Input: the truncation of h^{l1}: $trunc\_h$
Input: the truncation of s: $trunc\_s$
Initialize: $\mathbf{d}W^{\star}$, $\mathbf{d}b^{\star}$ $(\star = i, f, g, o, l2)$ to all zeros
$\mathbf{d}h^{l2} = \hat{y}_{(t)} - y_{(t)}$
$\mathbf{d}W^{l2} = \mathbf{d}h^{l2} \cdot h^{l1}_{(t)}$
$\mathbf{d}b^{l2} = squeeze(\mathbf{d}h^{l2})$
$\mathbf{d}h^{l1} = \mathbf{d}h^{l2} \cdot W^{l2}$
for $hs = t$ **to** $max(0, t - trunc\_h)$ **do**
    $\mathbf{d}s = \mathbf{d}h^{l1} \cdot o_{(hs)} \cdot tanh'(s_{(hs)})$
    $\mathbf{d}o = \mathbf{d}h^{l1} \cdot tanh(s_{(hs)})$
    $\mathbf{d}o\_input = \mathbf{d}o \cdot sigmoid'(o\_input_{(hs)})$
    $\mathbf{d}W^{o} += \mathbf{d}o\_input \cdot xc_{(hs)}$
    $\mathbf{d}b^{o} += squeeze(\mathbf{d}o\_input)$
    for $ss = hs$ **to** $max(0, hs - trunc\_s)$ **do**
        $\mathbf{d}g = \mathbf{d}s * i_{(ss)}$
        $\mathbf{d}i = \mathbf{d}s * g_{(ss)}$
        $\mathbf{d}f = \mathbf{d}s * s_{(ss-1)}$
        $\mathbf{d}g\_input = \mathbf{d}g * sigmoid'(g\_input_{(ss)})$
        $\mathbf{d}i\_input = \mathbf{d}i * sigmoid'(i\_input_{(ss)})$
        $\mathbf{d}f\_input = \mathbf{d}f * tanh'(f\_input_{(ss)})$
        $\mathbf{d}W^{\star} = \mathbf{d}\star\_input \cdot xc_{(ss)}$ $(\star = i, f, g)$
        $\mathbf{d}b^{\star} = squeeze(\mathbf{d}\star\_input)$ $(\star = i, f, g)$
        $\mathbf{d}s = \mathbf{d}s * f_{(ss)}$
        if $ss = hs$ **do**
            $\mathbf{d}xc = \sum^{\star=i,f,g,o} \mathbf{d}\star\_input \cdot W^{\star}$
            $\mathbf{d}h^{l1} = extract(\mathbf{d}xc)$
        endif
    end for
end for
Return: $NormalGrad = \{\mathbf{d}W^{\star}, \mathbf{d}b^{\star}\}$ $(\star = i, f, g, o, l2)$

```
16      return regularization
```

Listing 4.1: Implementation of regularization term by Anil Kag

It is worth attention that based on line 11 and line 15, the definition of $R_{(t)}$ is expressed as Exp.4.16.

$$R_{(t)} = -\theta * \lambda_{(t)} + \frac{\alpha}{2}||\theta - \overline{\theta}_{(t)}||^2 \tag{4.16}$$

By this, we have the corresponding $R'_{(t)}$ in Exp.4.17.

$$R'_{(t)} = \alpha(\theta - \overline{\theta}_{(t)}) - \lambda_{(t)} \tag{4.17}$$

So, it is conflicting that the coefficient of λ_t here in 4.17 is -1 rather than $-\frac{1}{2}$ we derived in 4.15. Note that *lambda* in the code is a hyperparameter, not λ_t. Since [2] implement the code to run experiments and report results, we stick to the code implementation and follow the gradient of $R_{(t)}$ in 4.17.

$$R'_{(t)}|_{\theta=\theta_{(t)}} = \alpha(\theta_{(t)} - \overline{\theta}_{(t)}) - \lambda_{(t)} \tag{4.18}$$

Parameter Update

With the gradients $NormalGrad$ and $R'_{(t)}$ that have been derived, we can then update the network parameter θ, the running average $\overline{\theta}$, and the running estimate λ in three sequential steps by Gradient Descent, as shown in Exp.4.19, Exp.4.20 and Exp.4.21. η is the learning rate.

$$\theta_{(t+1)} = \theta_{(t)} - \eta(NormalGrad + R'_{(t)}) \tag{4.19}$$

$$\lambda_{(t+1)} = \lambda_{(t)} - \alpha(\theta_{(t+1)} - \overline{\theta}_{(t)}) \tag{4.20}$$

$$\overline{\theta}_{(t+1)} = \frac{1}{2}(\overline{\theta}_{(t)} + \theta_{(t+1)}) - \frac{1}{2\alpha}\lambda_{(t+1)} \tag{4.21}$$

Note that the three expressions above follow the algorithm 2 in which partition factor K is set identical to T, so θ, $\overline{\theta}$, λ happen to share the same subscript (t) with R'. However, the subscript of θ, $\overline{\theta}$, λ can be positive integers no larger than K while that of R' can be positive integers no larger than T. In the next section, algorithm 4 presents a more general case.

4.1.2 Training Process Design

As all network components are available, we construct a computation flow of the training process by C programming so that we can perform profiling and understand the bottleneck. The flow is shown in algorithm 4 with scalable partition factor.

Algorithm 4 Training Process of S-MNIST with scalable partition factor

Input: Data samples of S-MNIST, $B = \{x^i, y^i\}_{i=1}^4$, Timesteps $T = 784$
Input: Learning rate η, Hyper-parameter α
Input: Partition factor set $KS = \{1, 2, 7, 14, 28, 56, 112, 392, 784\}$
Input: the truncation of h^{l1}: $trunc\_h$
Input: the truncation of s: $trunc\_s$
Initialize: $K \in KS$, $stride = \frac{T}{K}$
Initialize: θ_1 randomly in the domain θ
Initialize: $\overline{\theta_1} = \theta_1$ $\lambda_1 = 0$
Initialize: $h^{l1}_{(0)}$, $s_{(0)}$ to zeroes
for $p = 1$ **to** K **do**
    $subsequence = \{x^i\}_{i=1}^4$ **from** $(p-1) \cdot stride + 1$ **to** $p \cdot stride$
    $NetworkStates = \textbf{Forward}(\theta_p, subsequence, NetworkStates)$
    $NormalGrad$, $R'_{(t)} = \textbf{Backward}(\theta_p, \lambda_p, \overline{\theta_p}, \{y^i\}_{i=1}^4, NetworkStates, trunc\_h, trunc\_s)$ $(t = p \cdot stride)$
    $\theta_{(p+1)}, \lambda_{(p+1)}, \overline{\theta}_{(p+1)} = \textbf{ParameterUpdate}(NormalGrad, R'_{(t)}, \theta_{(p)}, \lambda_{(p)}, \overline{\theta}_{(p)})$ $(t = p \cdot stride)$
end for
Return: $\theta_{(K+1)}$, $CrossEntropy(\hat{y}_{(T)}, y_{(T)})$

To be specific, to train the LSTM network on the input sequence of length T by FPTT, the input sequence is partitioned into K parts/subsequences, and the length of every part is $stride$. For each part, we perform the

forward pass with the subsequence, generating *stride* number of network states. Then, the backward pass takes the generated states to calculate the gradients, which are used to update the parameter. A complete training process requires K iterations of such a routine.

Considering the memory, we visualize the difference between traditional one-go processing of sequence training and FPTT-based sequence partition in figure 4.1. It is clear that one-go processing requires $T+1$ memory block to store $T+1$ network states for the backward pass. However, due to sequence partition, the forward pass on the subsequence of length *stride* only has $stride+1$ network states to consume memory blocks. Furthermore, when the backward pass is finished on the network states, those memory blocks can be released and ready for processing of the next subsequence. In this way, we accomplish the training process by using $stride+1$ memory blocks for network states, reaching roughly K times saving compared with one-go processing.

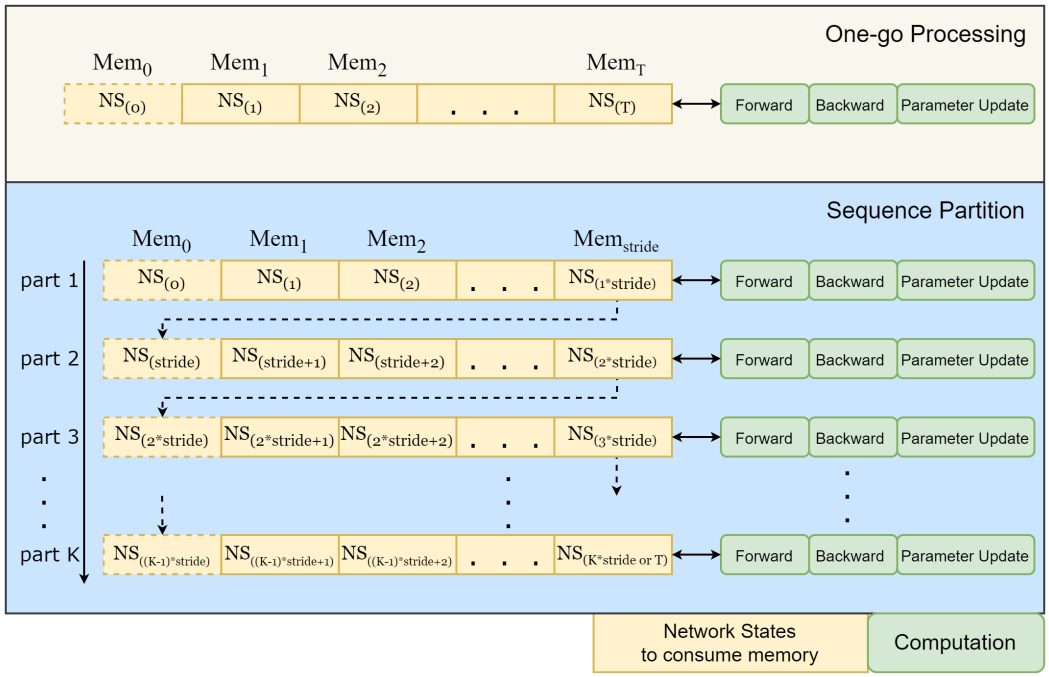

Figure 4.1: Visualization of the training process: One-go v.s. Sequence Partition

Table 4.2: Components of the workload

Type		Function Name
MatMul[1]	regular	mat_mul
	A transpose	mat_mul_a_T
	B transpose	mat_mul_b_T
EWO[2]	basic	element_wise_mac
		element_wise_mul
		element_wise_sub
	compound	find_ds
		find_d_l1_ifgo_input
	FPTT	FPTT_SGD
Activation	sigmoid[3]	sigmoid_on_matrix
	tanh[3]	tanhf_on_matrix
	softmax	softmax
Data Movement		load_sub_seq_to_xc
		relay_network_states
		fill_l1_h_into_xc
Matrix Squeeze		mat2vec_avr_squeeze

[1] Matrix Multiplications
[2] Element-wise Operations (linear) between matrices
[3] are also element-wise operations, though being non-linear

We implement the C program following the routine in algorithm 4. Furthermore, we also classify the components of the C program in table 4.2.

4.2 Results

4.2.1 Validation

We validate the correctness of the implemented LSTM-based network by checking its accuracy on the SMNIST dataset and compare the result with that of the same network in `PyTorch`.

As listed in table 4.3, our implementation can reach convergence with reasonable accuracy compared with the PyTorch version, though there is a 4% gap in streaming by rows and a 10% gap in streaming by pixel.

This difference may result from the way the backward pass is made. PyTorch automatically tracks operations happening on the tensor to compute gradients, which may be more precise than our analytic derivation.

Since convergence is reached, we suppose the amount of computation required is at a practical level for hardware acceleration design. In general, we consider that the implemented network is eligible for workload construction.

Table 4.3: Accuracies achieved by self-implemented LSTM network on MNIST

Network Version	streaming by rows	streaming by pixels
self-designed	94%	78%
PyTorch	98%	88%
Gap	4%	10%

4.2.2 Profiling

To understand the workload and find the bottleneck, we perform latency profiling on the workload of the training process.

We use `Linux gprof` as the profiling tool and list the result in table 4.4. It can be observed that Matrix Matrix Multiplication (MMM) is the top source of the latency and element-wise operations, mainly `FPTT_SGD`, i.e., parameter updates also become substantial when K gets larger. The results align with the figure 4.1 as parameter updates are as many as the K value.

Table 4.4: Percentages taken by most time-consuming function types for nine K values [1]

Function Type	K-001	K-002	K-007	K-014	K-028	K-056	K-112	K-392	K-784
MatMul	98.58	98.33	99.07	97.26	95.31	95.7	95.36	80.13	70.2
EWO & Activation	0.6	1.5	0.55	2.4	4.39	4.37	4.7	19.94	28.44
Others	<1								

[1] Due to the up rounding by `gprof`, the sum of components may be slightly greater than 100%

4.3 Discussion

This chapter is for Research Question 1. As FPTT has an affinity with the gate structure in RNNs, we propose the FPTT-based training process of the LSTM network. To be specific, we manually derive the forward pass, backward pass and parameter update of the LSTM with FPTT. Furthermore, we implement those components and construct the computation flow of the training process using C programming.

By analyzing and profiling the C program, we identify the dominance of MMM in computations. Therefore, an effective hardware solution for MMM would be the key to balancing latency and cost when accelerating FPTT for edge computing. On top of that, element-wise operations become non-trivial when the K value is large, which means we also need parallelism in the hardware solution.

Chapter 5

Hardware Acceleration Design

This chapter presents the methodology and results of hardware acceleration design for FPTT computations.

5.1 Methodology

The methods include how we propose, program, and optimize a scalable edge hardware architecture to efficiently accelerate the FPTT computations. First, we put forward the system architecture of one Matrix Multiplication accelerator with multiple general-purpose CPU cores. Meanwhile, we clarify our simulation-based HW/SW co-design flow. Then, for one thing, we introduce how to design C programs to manipulate the hardware architecture. For another, we perform fine-tuning and extension on the architecture for better efficiency and scalability.

5.1.1 System Proposal

We first present the motivation for the system architecture centering a Matrix Matrix Multiplication (MMM) accelerator, i.e., Gemmini, with multiple RISC-V cores named Rocket.

Choices

To begin with, we determine the basic type of the hardware accelerator for FPTT computation. Figure 2.3 in section 2.4.1 shows the trade-off between the application performance and the hardware flexibility. ASIC/hard-wired datapath has the highest efficiency, and many local learning algorithms are implemented as ASIC like E-Prop [1] and SDSP [29]. However, the Matrix Matrix Multiplication (MMM) in the FPTT computation flow dominates the latency as explained in table 4.4. In such a way, it is not suitable to be directly mapped to ASIC, as its dataflow is a series of hardly parallelizable matrix multiplications mixed with element-wise operations for which one hardware block of MMM can be expensive in area. Instead, it can be rewarding to reuse the MMM block across time to save the area while the performance can be barely influenced. Therefore, this leads to an ASIP architecture, i.e. Matrix Matrix Multiplication accelerator.

Admittedly, to cope with MMM, DSPs (or more generally Data Level Parallelism platforms) can also be a feasible solution as they offer parallel computational units to execute MMM efficiently. However, our program also contains operations like nonlinear activation functions and data scaling in the context of Artificial Neural Network (ANN). These computations are usually supported by the MMM accelerator for ANN directly at the hardware level but not on the DSP. Moreover, the DSP, usually as a commodity, is configured in a fixed way. At the same time, there are abundant open-source MMM accelerators for ANN which we can customize the scale of parallelism for our needs. Thus, considering the expertise and the scalability, the MMM accelerator is a more suitable solution than the DSP for our case.

Then, we investigate some MMM accelerators based on the feature of our workload. Our ideal accelerator is expected to support the RNN and has floating-point computation for precision-sensitive RNN training. Therefore, to the best of our knowledge, only Gemmini [30] meets the requirements though there are enormous open-source accelerators. Most solutions like NVDLA [1], Tensil AI [2] and Edge TPU [3] are either CNN-specific or do not support training. Gemmini also has a unique advantage in that it is available in the Chipyard [15], of which assets are introduced in section 2.5.1. In short, Chipyard makes it feasible to design and validate a complex system centering around Gemmini agilely.

[1] `http://nvdla.org/`
[2] `https://www.tensil.ai/`
[3] `https://cloud.google.com/edge-tpu`

To summarize, we choose Gemmini, a MMM accelerator as the backbone of the hardware architecture for FPTT acceleration, considering efficiency, flexibility, expertise and scalability.

One Gemmini, Multiple Rocket

Furthermore, we propose and explain the complete system architecture of one Gemmini and multiple Rocket cores, as shown in figure 5.1 which is an instance of the template in figure 2.4.

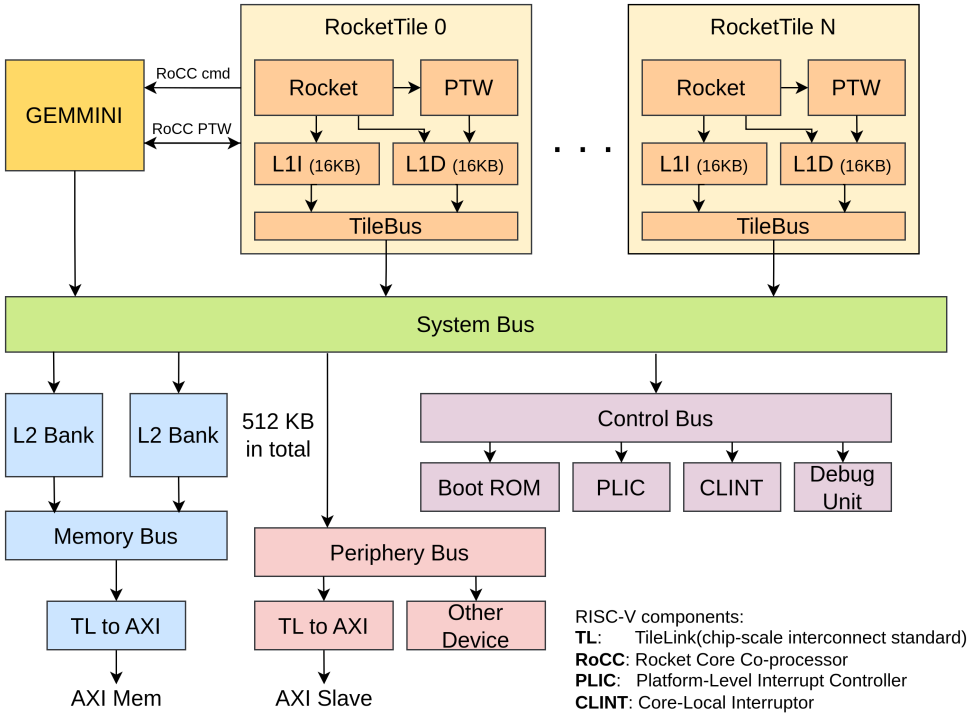

Figure 5.1: Architecture: One Gemmini, Multiple Rocket (adapted from the source)

We use Gemmini, the systolic array-based accelerator for Matrix Matrix Multiplication (MMM), to accelerate both the MMM and activation functions. Figure 5.2 presents the details of Gemmini Architecture. In general, Gemmini is a RoCC accelerator that interacts with external blocks for custom RISC-V ISA commands, PTW and data in the L2 cache.

Gemmini comprises a controller, scratchpad, matrix transposer, systolic array and other periphery functions. Implementation of Gemmini follows the idea of access-execute decoupling, and it has detached `Load`, `Store` and `Execute` instructions. The dependency management unit (or Re-Order Buffer, ROB) solves the instruction dependency and improves instruction-level parallelism. DMA engine transfers data between main memory and scratchpad/accumulator. Local TLB collaborates with the PTW in the Rocket Tile to keep the address mapping.

The scratchpad is composed of multiple banks of SRAM, and it buffers the input and output data to the systolic array. The transposer can transpose the matrix before it enters the systolic array.

Figure 5.3 explains the 2-level hierarchy of the systolic array. It is a mesh of tiles, between tiles the register is inserted. Inside a tile is a fully combinational array of PE (Processing Element). Each PE can be set to two dataflows: weight stationary and output stationary. The difference between these two is that either the weight (one source matrix) or the accumulator (the bias) is preloaded into the register in the PE. We can choose one dataflow when we configure and generate the hardware or instantiate both and leave the decision in the program.

Periphery functions include activations, scaling, and Accumulator SRAM. ReLU, ReLU6, Layer Normalization, and Softmax are the supported activations. Scaling multiplies the data with a scaling factor when the data enters or leaves the systolic array. Accumulator SRAM is another bank of SRAM used to store the GEMM accumulator in weight stationary mode.

Additionally, we instantiate multiple RISC-V cores to accelerate Element-wise operations that are another type of bottleneck in the FPTT computation shown in table 4.4. In principle, two options exist for such operations within the Chipyard framework. The first is a vector coprocessor Hwacha [31]. It could be a perfect solution for element-wise operations. However, it is hardly feasible for three reasons. Hwacha was developed early in RISC-V ISA history and uses a non-standard RISC-V extension (XHwacha) rather than the RISC-V standard vector extension proposal. Moreover, because of the non-standard extension, a separate set of binary

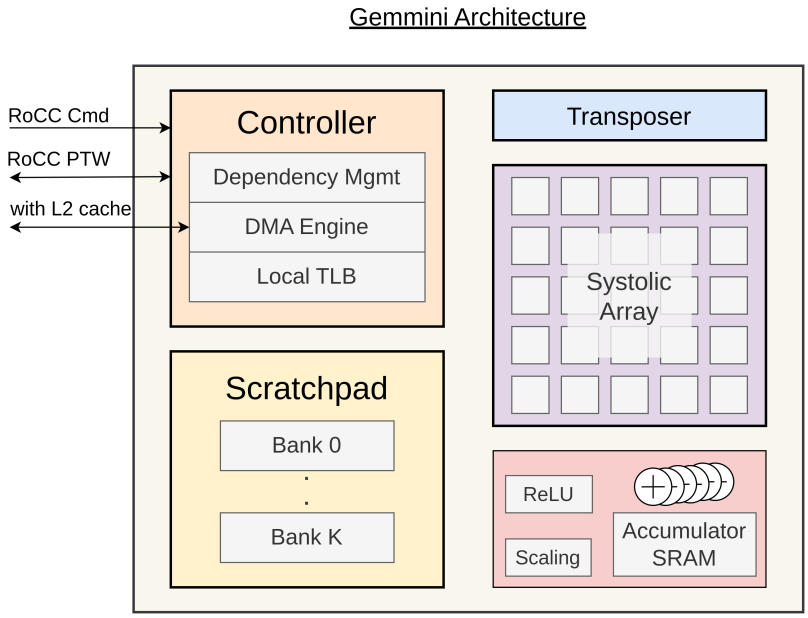

Figure 5.2: Architecture of Gemmini Accelerator (source)

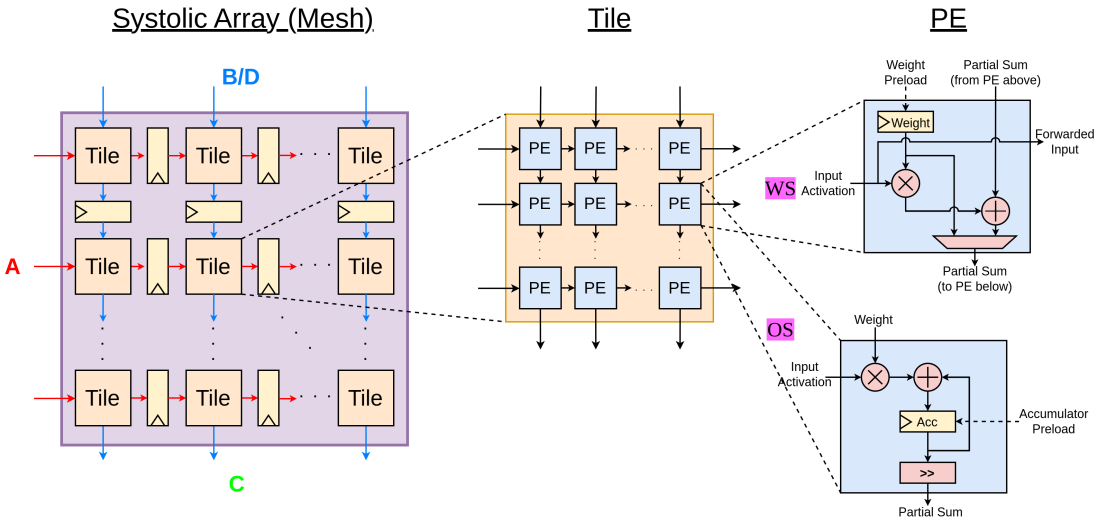

Figure 5.3: Structure of the systolic array in Gemmini (source)

toolchains is needed to process the program, which is not being maintained promptly. Furthermore, compiler support is relatively limited, so programmers must design assembly code to manipulate the hardware, making it hard to program and debug.

So, we opt for the multicore system by attaching multiple `RocketTile` to the System Bus to parallelize element-wise operations. Cache coherency of the multicore system is also handled by `TileLink` protocol. With this option, we do not need to switch to another infrequently maintained binary toolset, and we can reuse the C code of the single-core system with minor adaptions.

Table 5.1 lists all RISC-V cores available in the Chipyard. Based on the property of our workload, floating point operations (F) are required in the architecture, so the candidates include Rocket (Big), BOOM, and Shuttle RISC-V. However, shuttle RISC-V can be configured with up to four cores. Then, it is not considered because of the limited scalability. Also, BOOM is an out-of-order machine aiming at high-performance circumstances, which may not be preferable compared to the in-order Rocket core in embedded computing. Therefore, Rocket (Big) is chosen to construct the system.

In summary, we are motivating a system-level solution of one Gemmini and multiple Rocket to accelerate Matrix Matrix Multiplication and element-wise operations in the Forward Propagation Through Time (FPTT) computations.

Table 5.1: RISC-V cores in Chipyard

Name		ISA
Rocket	big	RV64IMAFDC
	medium	RV64IMAC
	small	RV64IMAC
BOOM		RV64IMAFDC
CVA6 (Ariana)		RV64IMAC
Ibex		RV32IMC
Sodor		RV32IM
Shuttle RISC-V		RV64IMAFDC

5.1.2 HW/SW Co-Design Flow

The HW/SW co-design flow explains, at a high level, how we coordinate hardware customization and software adaptation to approach an RNN-efficient and scalable acceleration solution, while details of HW/SW change are in the following section 5.1.3 and 5.1.4. The method is simulation-based, and three ways of simulations applied are elaborated in section 2.5.2. Figure 5.4 demonstrates the three-step methodology co-design flow, and the background color at each step represents the simulation ways available.

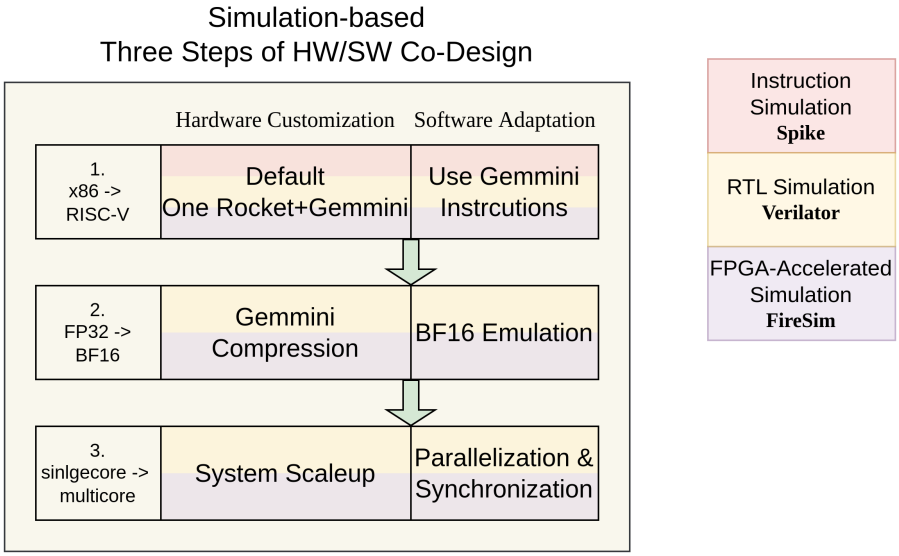

Figure 5.4: The flow of HW/SW co-design

As a starting point, we use the default combination of Rocket and Gemmini for the hardware and perform architecture relocation on the software from x86 to RISC-V. Adaptations cover RISC-V/Gemmini-specific C libraries and binary toolsets. At this step, we debug the code by instruction simulation first to ensure correctness. Then we can obtain the performance profiling by FPGA-Accelerated Simulation, i.e. `FireSim`.

Next, we perform architecture compression on Gemmini, including switching to BF16 calculations from FP32. To that end, the key software modification is BF16 emulation on the Rocket core that does not directly support BF16. Furthermore, since `Spike` does not support BF16, we have to guarantee the correctness of the BF16 code using the slow RTL simulator by checking the individual functions one by one. Then, similarly, we use `FireSim` to test and profile the full workload.

Last, using the compressed Gemmini architecture, we further scale up the full system by utilizing more than one core to inspect the efficiency of the multicore system, as well as a larger systolic array in Gemmini. To that end, we add the mechanism of parallel execution and synchronization in the C code. `FireSim` is also applied to this step.

In general, `FireSim` serves as the core of our method as it provides the resource utilization of the design point as the hardware cost, together with the cycle-accurate performance profiling. Appendix E shows how we establish the `FireSim` platform.

5.1.3 Programming Method

We then introduce the programming method of the system, by which we can perform the software adaptation in the three-step co-design flow.

Gemmini

The Gemmini is compatible with high-level, mid-level and low-level programming for users needing different levels of manipulation.

High-level programming uses ONNX (Open Neural Network Exchange [4]), which is an open format built to represent machine learning models. Most deep learning frameworks like PyTorch and Tensorflow can export models in ONNX format. To be specific, the Chipyard team forked `ONNX Runtime`, the mapper that distributes the ONNX workload on specific hardware, to enable ONNX models to run on Gemmini. Mid-level programming is to utilize a hand-tuned C library of kernels. The library covers most functions in the domain of deep learning, like matrix multiplication and max-pooling. It is also critical to make sure that a C header file describing the architecture configuration matches the real hardware structure. Low-level programming is to program in assembly instructions, which is the way to design mid-level kernels and conduct performance tuning, rather than an efficient method of application development.

We opt for mid-level programming because the `ONNX Runtime` for high-level programming supports FP32 only at the moment, while Brain Float 16 is an option that is worth exploration as mentioned in section 2.4.2. Afterward, to utilize Gemmini for acceleration by C programming, it is substantial to understand the interface of `tiled_matmul_auto` function in the Gemmini C library. The function automatically tiles the source matrices and performs the calculation and assembles the results. Table 5.2 lists the arguments.

Table 5.2: Argument list of `tiled_matmul_auto`

Division	Name	Data Type
Dimensions	dim_I	`size_t`
	dim_J	`size_t`
	dim_K	`size_t`
Matrix Address	A	`const elem_t *`
	B	`const elem_t *`
	D	`const void *`
	C	`void *`
Matrix Stride	stride_A	`size_t`
	stride_B	`size_t`
	stride_D	`size_t`
	stride_C	`size_t`
Scaling Factor	A_scale_factor	`scale_t`
	B_scale_factor	`scale_t`
	D_scale_factor	`scale_acc_t`
	scale	`acc_scale_t`
Activation	act	`int`
Transpose	transpose_A	`bool`
	transpose_B	`bool`
Precision	full_C	`bool`
	low_D	`bool`
Bias option	repeating_bias	`bool`
Dataflow	tiled_matmul_type	`enum tiled_matmul_type_t`

`tiled_matmul_auto` implements the matrix operation defined in Exp 5.1. Matrix A is multiplied by B. The result of the multiplication is added by D, and then a non-linear activation function σ is applied to the sum. Moreover, the final result is assigned to matrix C. In the context of neural networks, A and B are input activation and weight, D is the bias, σ is the activation function and C is the output.

$$C = \sigma(A \cdot B + D) \tag{5.1}$$

dim_I and dim_J are the row and the column dimension of the matrix C. dim_K is the remaining shared dimension by A and B. Arguments A, B, C and D define the source address of the array that stores the matrix. The four following arguments, i.e. stride_* (*=A, B, D, C), represent the column dimension of the

[4] `https://onnx.ai/`

corresponding matrix. By four scaling factors, we can apply data scaling on the target matrix. For example, when we want to average the matrix C by the batch size, we can set the argument scale to $\frac{1}{batch\_size}$. With regard to the activation function, values 0 to 4 stand for no activation, ReLU, layer normalization, IGELU, and softmax respectively. On top of that, we can choose whether to transpose the matrix A and B, and whether to use full or low precision of matrix C and D. If the size of D is just one row of C, repeating_bias should be set to repeat the bias on all rows of C. Last, two dataflows are available, i.e. WS and OS.

Multiple Rocket

Furthermore parallel acceleration, multicore synchronization, and BF16 emulation are the key considerations for Rocket cores.

In the context of RISC-V ISA, `Hart` (or Hardware Thread) represents an independent processing unit or a core. Then, each core or `Hart` in the system will be assigned a unique `HartID`, and we can allocate different software threads to the expected `Hart` by matching the `HartID`. On top of that, we use the `barrier` for thread synchronization. An example program is given in listing 5.1 that implements the element-wise addition of arrays (or Vector-Vector Addition) on a multicore platform. The key part is on line 34 where each core works on the data of different indexes in the array in parallel. Afterward, all threads are synchronized by calling the barrier defined on line 6.

```
1  #include <riscv-pk/encoding.h>
2  #include <stdio.h>
3  #define N 10 // size of vectors
4  static size_t n_cores = 4; // the number of cores to be used
5
6  static void __attribute__((noinline)) barrier() // definition of barrier
7  {
8    static volatile int sense;
9    static volatile int count;
10   static __thread int threadsense;
11
12   __sync_synchronize();
13
14   threadsense = !threadsense;
15   if (__sync_fetch_and_add(&count, 1) == n_cores-1)
16   {
17     count = 0;
18     sense = threadsense;
19   }
20   else while(sense != threadsense);
21
22   __sync_synchronize();
23 }
24
25 void __main(void) {
26   size_t mhartid = read_csr(mhartid);
27   int x[N], y[N], z[N];
28
29   // initialize arrays
30   // ...
31
32   if (mhartid >= n_cores) while (1);
33
34    for (size_t i = mhartid; i < N; i+=n_cores)
35    {
36       z[i] = x[i] + y[i];
37    }
38    barrier();
39
40   if (mhartid > 0) while (1); // Spin if not core 0
41 }
42
43 int main(void) {
44   __main();
45   return 0;
46 }
```

Listing 5.1: Example program for the multicore platform

Besides the parallel execution and synchronization of Element-wise operations, it is also critical for the Rocket core to process data in BF16. Unfortunately, Rocket does not support direct BF16 calculations, however, BF16 and FP32 can be converted to each other by bit shifting.

Specifically, we store BF16 data in type `uint16_t` in a C program. Before doing any calculations, we convert the data to FP32 so that the CPU can handle it as FP32 data. After the FP32 operation by the Rocket, we convert the result in FP32 back to `uint16_t`. Macros are in list 5.2. Note that such conversions can induce nontrivial overhead.

```
1 #define u16_to_u32(dat) ((uint32_t)dat << 16)
2 #define u32_to_fp32(dat) (*(float*)&dat)
3
4 #define fp32_to_u32(dat) (*(uint32_t*)&dat)
```

```
#define fp32_to_u16(dat) ( (fp32_to_u32(dat)>>16) + ((fp32_to_u32(dat)&0x8000)==0x8000) ) // up rounding
```

Listing 5.2: Mutual conversion between FP32 and BF16

5.1.4 System Customization

System customization for FPTT computation has two steps. First, we perform Gemmini architecture compression in terms of quantization and reduction. Specifically, we use a smaller but also efficient datatype, i.e. Brain Float 16 instead of 32-bit Float, and remove unnecessary hardware blocks for the workload to lower the resource utilization while maintaining latency and accuracy. Because this is not extensive DSE as FPGA-based synthesis is time-consuming, we do not explore most quantified parameters like the size of TLB. Second, we explore the system dimension, i.e. systolic array size and the number of Rocket cores, since they are the main contributor to performance and resource utilization.

Gemmini Compression

In the first step of architecture compression, we change part of the Gemmini parameters as shown in table 5.3, and it is recommended to read the full list of parameters in Appendix B first.

Table 5.3: Gemmini architecture compression

Division	Parameter	Default		Custom BF16
		FP32	BF16	
Arithmetic	inputType	Float(8,24)	Float(8,8)	Float(8,8)
	spatialArrayOutputType	Float(8,24)	Float(8,8)	Float(8,8)
	accType	Float(8,24)		Float(8,8)
Controllers	ex_read_from_acc	true		false
	ex_write_to_spad	true		false
Data Scaling	mvin_scale_args.multiplicand_t	Float(8,24)		Float(8,8)
	mvin_scale_acc_args.multiplicand_t	Float(8,24)		Float(8,8)
	acc_scale_args.multiplicand_t	Float(8,24)		Float(8,8)
Systolic Array	dataflow	Dataflow.BOTH		Dataflow.WS
Scratchpad	sp_capacity	256 KB		128 KB
Accumulator	acc_capacity	64 KB		32 KB
Periphery	has_max_pool	true		false
	has_dw_convs	true		false
	has_first_layer_optimizations	true		false

The floating point Gemmini default parameter setting provides options FP32 and BF16, but they are not optimal for FPTT computations. Therefore, we propose a custom BF16 based on the default settings.

For arithmetic, in section 2.4.2, Kalamkar et al. [14] points out that BF16 can achieve the same state-of-the-art results as FP32. So, BF16 is comprehensively applied on top of default BF16, not only for input and output type but also for `accType`, i.e. the type of bias in matrix multiplication for accumulation. Also, `accType` being FP32 results in a datatype casting error, which is explained in Appendix D. Furthermore, we set all types of scaling factors to BF16. Overall, these changes lead to the global application of BF16, which can lower the computation complexity and resource utilization to a greater extent, though it may induce a small accuracy loss, compared with the mixed precision of FP32 and BF16.

In addition, we do not instantiate both dataflow modes in the systolic array to reduce resource utilization. Instead, we only use Weight Stationary (WS) rather than Output Stationary (OS), since Gookyi et al. [32] report that the WS dataflow offered a speedup of 3× relative to the OS dataflow.

In terms of memory, we halve the size of both the Scratchpad and Accumulator, as we have chosen the BF16, the bit width of which is half of that of the FP32. In the periphery, we ignore the hardwired optimization that is irrelevant to our workload. We also deactivate the unnecessary data path in the controller to save the area.

System Scaleup

Subsequently, based on the custom BF16 Gemmini by architecture compression, we further investigate the scalability of the system, as well as the trade-off of the proposed architecture between FPTT computation performance in latency and hardware cost in resource utilization. To be specific, we tweak the array size and the number of Rocket cores.

Parameters determining the array size are in table 5.4. The systolic array, or the mesh, is a cluster of tiles, with registers between tiles. `tileRows` × `tileColumns` defines the number of PE in a tile, and `meshRows` ×

`meshColumns` is the number of tiles in the mesh/array. The row and the column are required to be identical. The tile is fully combinational, so usually there is only one PE in it to avoid long combinational paths by setting both `tileRows` and `tileColumns` to 1. `meshRows` and `meshColumns` need to be the power of 2, starting from 4. Therefore, we explore the effect of mesh size within {4,8}. Note that 16x16 mesh is not examined as it can hardly fit in the FPGA board.

Table 5.4: Parameters of Systolic Array Size

Name	Default Value	Set
tileRows	1	1
tileColumns		
meshRows	4	{4,8}
meshColumns		

Meanwhile, deployment of huge designs in the FPGA at a reasonable frequency for simulation can be challenging; we slightly give in to the latency of the systolic array by inserting more registers to facilitate the placement and routing. Five parameters of latency are incremented, as shown in the table 5.5.

Table 5.5: Parameters of Gemmini relating to the timing of FPGA implementation

Division	Parameter	Default	Set
Systolic Array	tile_latency	2	4
	mesh_output_delay	0	2
Data Scaling	mvin_scale_args.Latency	4	6
	mvin_scale_acc_args.Latency	4	6
	acc_scale_args.Latency	4	6

For the number of Rocket, we have {1,2,4,6,8,10}. Since the dimensions of most matrices in the workload are even numbers, we also have the number of cores in even numbers except in the basic single-core case.

With two systolic array sizes and six candidate core numbers in the system, we list 12 design points in table 5.6.

Table 5.6: Design points of the system

Index	$SIZE_{mesh} - N_{cores}$
1	4x4-1
2	4x4-2
3	4x4-4
4	4x4-6
5	4x4-8
6	4x4-10
7	8x8-1
8	8x8-2
9	8x8-4
10	8x8-6
11	8x8-8
12	8x8-10

5.2 Results

This section presents the results of Gemmini architecture compression and system scaleup.

5.2.1 Gemmini Compression

We first demonstrate the efficacy of architecture compression by listing the comparisons between the default-FP32 Gemmini and our custom-BF16 Gemmini in terms of resource utilization and the profiling of the software workload on architectures, including loss function trend, latency, and memory consumption.

The workload we used is explained in section 4.1.2, for which nine options of K value, by FPTT algorithm, are available within {1,2,4,7,28,56,112,392,784}. Thus, in parallel to the effect of architecture compression, we also exhibit the property of sequence partition in FPTT theory, which is a trade-off between the frequency of parameter updates and the memory footprint to process the sequence.

Resource utlization

Table 5.7 is the comparison of resource utilization. The decrease in `LUT`, `BRAM` and `DSP` are roughly 50%, while the `Register` experiences 20% drop. It is also worth mentioning that unlike FP32, the Multiply and Accumulate (MAC) units in the BF16 systolic array are constructed by Configurable Logic Block (CLB) only, i.e. no DSPs are consumed. This causes a reasonable decrease in DSP usage, while DSPs still existing in custom-BF16 architecture are for address calculation in the controllers.

In general, architecture compression brings about significant savings in resource utilization.

Table 5.7: Comparison of Gemmini Resource Utilization

Architecture	LUT	Register	BRAM	DSP
default-FP32	94969	31067	80	256
custom-BF16	42022	23818	40	120
change	-56%	-23%	-50%	-61%

Precision

Table 5.8 compares the Cross-Entropy Loss at the final time step of the workloads of nine K values, i.e., the partition factor, with two precision FP32 and BF16. BF16 can result in similar trainability as FP32 does, though FP32 usually achieves slightly better under the same condition.

Table 5.8: Comparison of Loss Function at final time step

Type	K-001	K-002	K-007	K-014	K-028	K-056	K-112	K-392	K-784
FP32	0.058755	0.058257	0.057592	0.056168	0.053817	0.051874	0.044308	0.027446	0.019935
BF16	0.071089	0.062993	0.070962	0.055088	0.079186	0.047183	0.071025	0.023592	0.023592

Furthermore, figure 5.5 depicts the Cross-Entropy Loss Trend by both precisions when K is set to 784. For an S-MNIST sequence with 784 time steps, i.e., $T = 784$, K also being 784 means that the backward pass and weight update would be done for every time step. Therefore, that leads to most calculations among the workload of nine K options and is consequently considered most precision-sensitive.

It can be observed that BF16 provides a very close trend to what FP32 can do, especially after two-thirds of the time steps are provided, i.e. when the shape of the digit is completely "visible". As a result, BF16 is comparable to FP32 on our workload.

Note that the loss is sampled every 28 time steps rather than all 784 steps. Since the MNIST figure is a 28x28 2D pixel array with a digit in the center, the loss function becomes unstable when a row is not presented entirely. So, to have a reasonable plot, we sample the loss at the end of every row or every 28 time steps.

Matrix Multication Latency

Figure 5.6 compares the cumulative latency of matrix multiplications in the workload on two architectures. It is clear that the latency by custom-BF16 is slightly smaller than that of default-FP32. Both architectures are configured with identical latency in computation blocks, so we suppose the improvements stem from data transfer latency of less width.

On top of that, there is a gradual decline in the matrix multiplication latency when K gets larger. Since the subsequence length is shorter for larger K, the backpropagation of $s^{(t)}$ inside the subsequence is forced to be truncated earlier than it is set to. This is reflected in line 15 of Algorithm 3 when $hs - trunc\_s$ is smaller than 0.

Total Latency

Although matrix multiplication latency drops, the total latency goes up on custom-BF16 as shown in figure 5.7, because of the conversion overhead between FP32 nad BF16 by Rocket core mentioned in section 5.1.3. The bar of total latency is a stack of functions listed in the right legend, from the bottom up. It can be noticed that

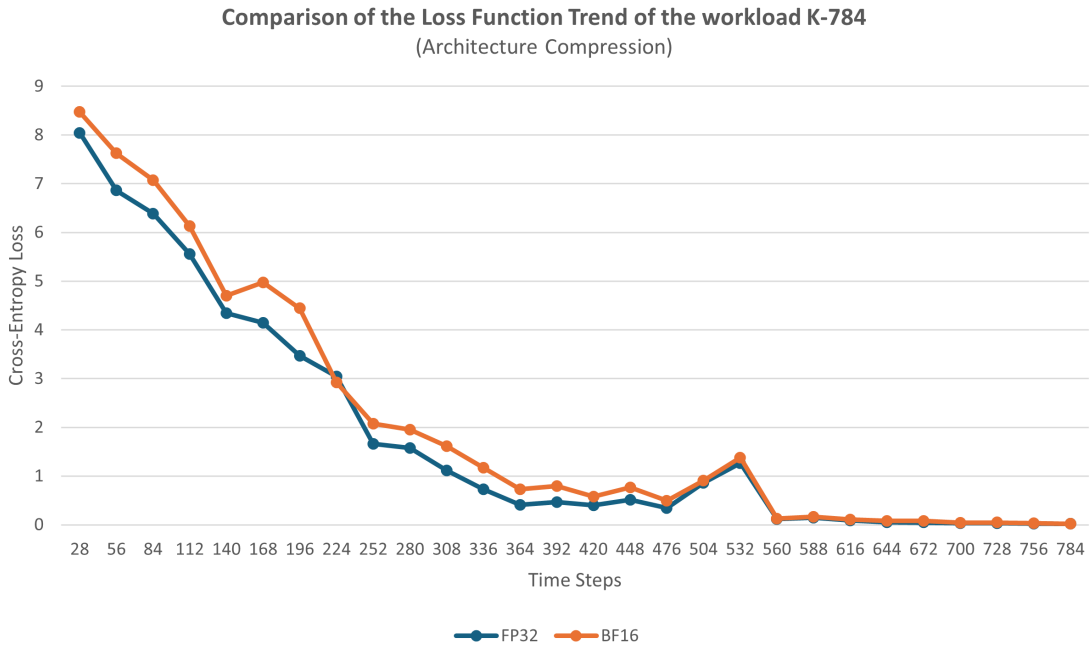

Figure 5.5: Comparison of Loss Function Trend of the workload K-784

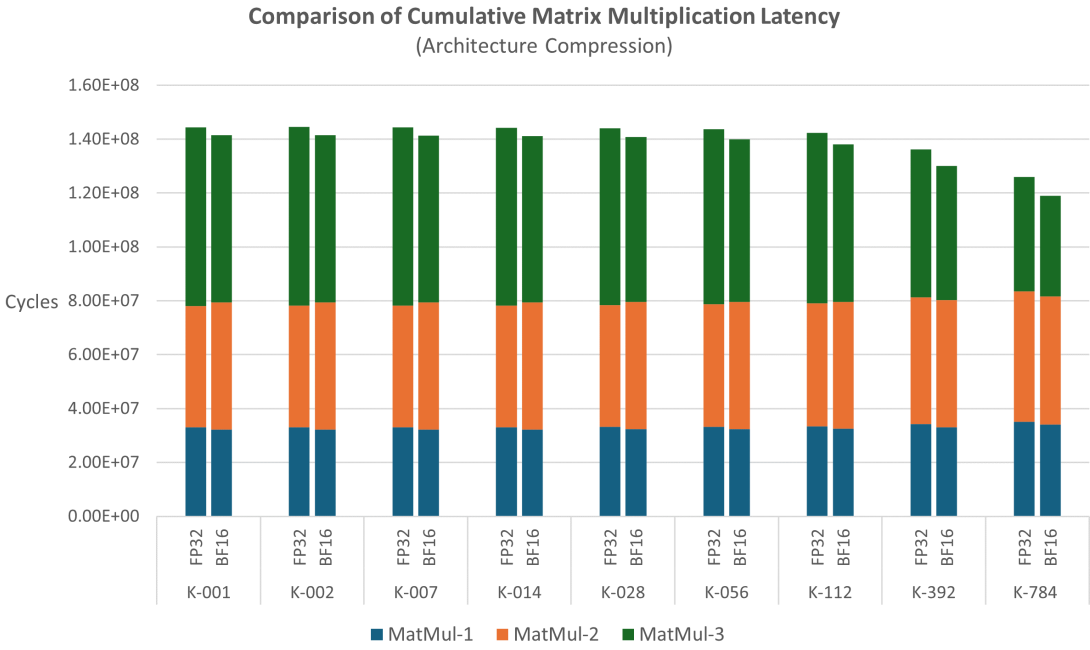

Figure 5.6: Comparison of Cumulative Matrix Multiplication Latency

the latency of functions mapped on the CPU, like `EWO-SGD(FPTT)`, `EWO-combo` and `EWO-basic`, almost doubles from FP32 to BF16 at each K value. The overhead is even more serious for the workload with large K, in which EWO is dominant. Conversion overhead is the downside of using BF16, due to the incapability of BF16 calculation of the CPU at the moment.

Memory Usage

Figure 5.8 shows the static memory usage simply halved for each K when we switch to BF16 from FP32.

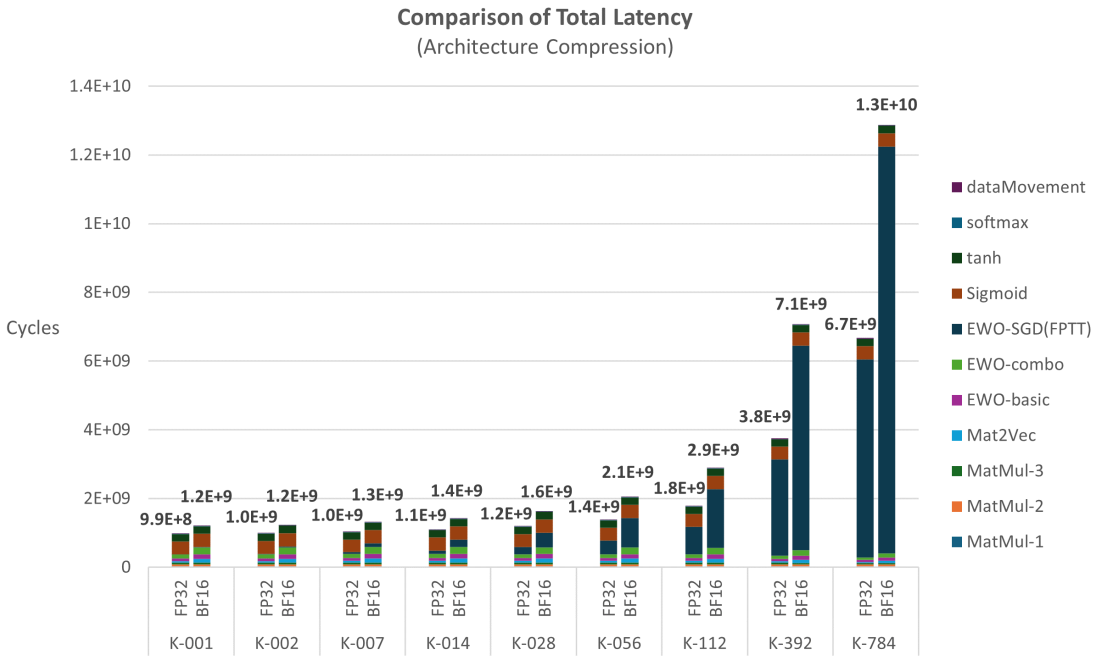

Figure 5.7: Comparison of Total Latency

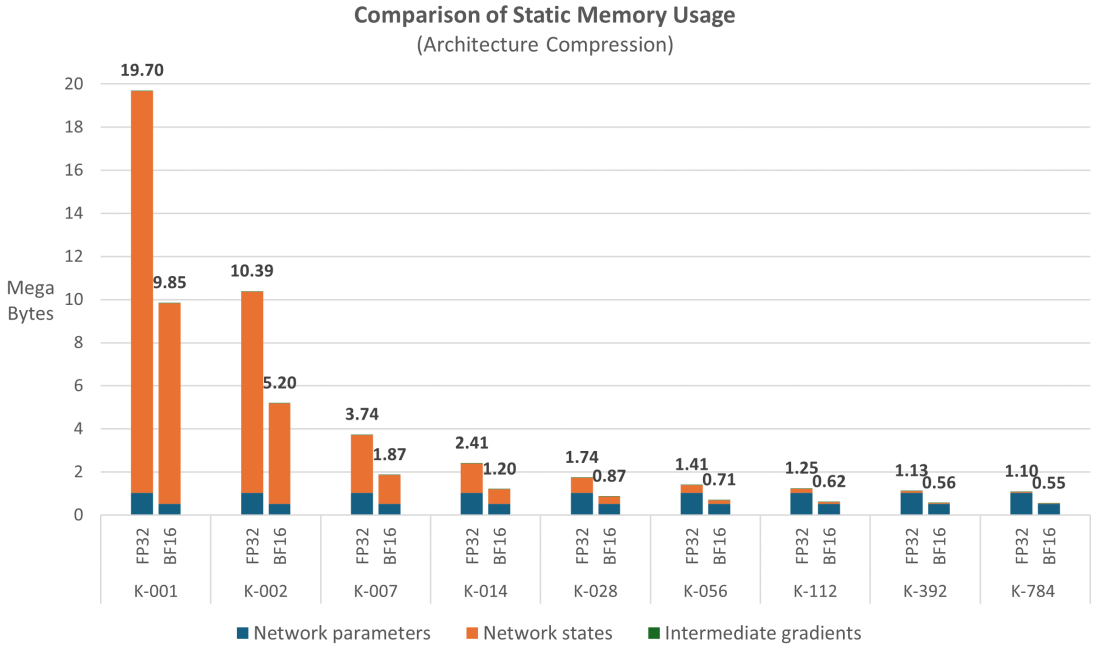

Figure 5.8: Comparison of Static Memory Usage

Trade-off of Sequence Partition

Apart from the effect of BF16 on the total latency and memory usage, figure 5.7 and figure 5.8 also demonstrate the trade-off provided by the partition factor K in FPTT theory, i.e. between the network parameter updates and memory footprint.

Large K means more frequent parameter updates so more executions of the optimizer routine described in section 4.1.1 or `EWO-SGD(FPTT)`, which leads to the upgoing trend in figure 5.7.

On the other hand, a large K means the input sequence is partitioned into more subsequences of shorter length. Therefore, fewer network states are generated for a subsequence, and fewer memory blocks are consumed for storing network states for the backward pass. Nevertheless, the network parameters and intermediate gradients stay the same for all K values as they are irrelevant to K.

K being close to 28 (or $\sqrt{784}$) appears to be the optimal partition factor that balances latency and memory

footprint, from figure 5.7 and figure 5.8. K being 28 with BF16 brings the memory footprint down to 1 MB, and further partitioning can not make significant gains in memory for embedded devices while the latency starts to go up substantially. In general, the partition of 28 together with BF16 can lead to $\frac{19.7}{0.87} \approx 22.6$ times save in memory footprint compared with $K-001$, at the cost of $\frac{1.6e9}{1.2e9} \approx 1.33$ times latency. Note that $K-001$ means no sequence partition, so its memory footprint is close to BPTT training, while the only difference is network parameters include $\overline{\theta}$ and λ.

Summary

In summary, architecture compression is beneficial. In terms of hardware, there are considerable savings in resource utilization. With regard to the profiling of the workload, BF16 has a learning curve similar to that of FP32 on our workload; BF16 slightly shortens the latency of data transfer; BF16 halves memory usage compared with FP32. Admittedly, conversion overheads are nontrivial for the CPU, but they are with element-wise operations and subsequently can be alleviated by parallel execution on a multicore platform.

5.2.2 System Scaleup

Next, we move on to the results of the system scaleup.

Resource Utilization

Table 5.9 and 5.10 lists how the resource utilization changes with the system scaling up, for Gemmini architecture only and the entire system.

As in Table 5.9, significant growths are seen on `LUT` and `Register` when we increase the mesh size. Since the Scratchpad and Accumulator sizes stay the same, BRAM remains unchanged. Similarly, DSP usage is not growing as the computing unit in the PE is synthesized to `LUT` and `Register`, i.e. not using DSP.

Table 5.9: Comparison of Gemmini Resource Utilization

custom-BF16-Gemmini	LUT	Register	BRAM	DSP
4x4	42777	24229	40	120
8x8	81106	34551	40	120
change	+90%	+43%	0%	0%

In terms of the full system in table 5.10, where resource utilization of design point 4x4-1 is considered baseline, the usage of `LUT` and `Register` roughly quadruples with the system expanding to 10 cores, while the other two resource doubles. The extra `BRAM` is mainly consumed as the 16KB data cache and 16KB instruction cache in the added Rocket core, while `DSP` is for the FPU inside the Rocket cores.

Table 5.10: Resource Utilization of Design Points

$SIZE_{mesh}$-N_{core}	LUT		Register		BRAM		DSP	
	Abs	Increase	Abs	Increase	Abs	Increase	Abs	Increase
4x4-1	86235	**1.0**	46766	**1.0**	168	**1.0**	135	**1.0**
4x4-2	115257	**1.3**	60224	**1.3**	180	**1.1**	150	**1.1**
4x4-4	172876	**2.0**	87117	**1.9**	196	**1.2**	180	**1.3**
4x4-6	230855	**2.7**	114008	**2.4**	212	**1.3**	210	**1.6**
4x4-8	288814	**3.3**	140893	**3.0**	228	**1.4**	240	**1.8**
4x4-10	346827	**4.0**	167839	**3.6**	244	**1.5**	270	**2.0**
8x8-1	123864	**1.4**	61105	**1.3**	168	**1.0**	135	**1.0**
8x8-2	152860	**1.8**	74561	**1.6**	180	**1.1**	150	**1.1**
8x8-4	210569	**2.4**	101454	**2.2**	196	**1.2**	180	**1.3**
8x8-6	268190	**3.1**	128347	**2.7**	212	**1.3**	210	**1.6**
8x8-8	326148	**3.8**	155229	**3.3**	228	**1.4**	240	**1.8**
8x8-10	384080	**4.5**	182179	**3.9**	244	**1.5**	270	**2.0**

Total Latency

Table 5.11 includes the workload latency on the system of 12 design points. We assume 500HMz as the frequency, according to products of similar specifications presented in section 3.3, though on `FireSim` simulation we have tens of megahertz.

It is noticeable that, regardless of whether it is for a 4x4 or 8x8 mesh, the system with 6 or more Rocket cores can process the workload of relatively small K values within 1 second. Therefore, it would be possible to finish a complete training script assuming 30 iterations within a half-minute.

It is worth mentioning that a larger mesh only negligibly improves latency for our workload, as a 4x4 mesh can already squeeze the matrix multiplication to a minor fraction of the workload.

Table 5.11: Workload Latency in seconds by design points (assuming 500MHz)

$SIZE_{mesh}$-N_{core}	K-001	K-002	K-007	K-014	K-028	K-056	K-112	K-392	K-784	$\frac{K-028}{K-001}$
4x4-1	2.47	2.50	2.68	2.89	3.30	4.15	5.82	14.17	25.77	1.34
4x4-2	1.71	1.74	1.88	2.05	2.38	2.66	3.60	8.02	13.96	1.39
4x4-4	1.15	1.16	1.21	1.27	1.39	1.66	2.13	4.65	7.86	1.21
4x4-6	0.91	0.91	0.95	0.99	1.07	1.28	1.59	3.30	5.57	1.18
4x4-8	0.80	0.80	0.83	0.86	0.93	1.11	1.33	2.70	4.46	1.16
4x4-10	0.73	0.74	0.76	0.79	0.84	1.01	1.18	2.33	3.84	1.15
8x8-1	2.32	2.36	2.54	2.75	3.16	4.00	5.68	14.05	25.66	1.36
8x8-2	1.57	1.60	1.72	1.90	2.23	2.52	3.47	7.89	13.84	1.41
8x8-4	1.00	1.01	1.06	1.12	1.24	1.52	1.99	4.51	7.74	1.23
8x8-6	0.76	0.77	0.80	0.84	0.93	1.14	1.45	3.18	5.46	1.22
8x8-8	0.66	0.66	0.69	0.72	0.79	0.97	1.20	2.58	4.37	1.20
8x8-10	0.59	0.60	0.62	0.65	0.70	0.86	1.05	2.20	3.72	1.19
Max Speedup	4.19	4.20	4.34	4.47	4.70	4.80	5.55	6.45	6.93	-

Table 5.12: Reference Workload Latency in seconds on alternative platforms

Platform		K-001	K-002	K-007	K-014	K-028	K-056	K-112	K-392	K-784
single Rocket core (500MHz)		22.06	22.08	22.12	22.70	22.85	22.62	23.74	25.94	29.64
NVDIA GPU	T4 [1]	2.83	2.83	2.83	2.86	2.90	2.97	3.14	3.91	5.04
	L4 [2]	2.85	2.87	2.88	2.91	2.95	3.03	3.18	4.00	5.12
	V100 [3]	2.91	2.92	2.98	2.99	2.98	3.10	3.18	4.04	5.18
Intel Xeon (2.2GHz)		0.61	0.72	1.23	0.53	0.54	0.59	0.66	0.94	1.34

[1] `https://www.nvidia.com/en-us/data-center/tesla-t4/`
[2] `https://www.nvidia.com/en-us/data-center/l4/`
[3] `https://www.nvidia.com/en-us/data-center/v100/`

For reference, we run the workload on other platforms. The first is the single Rocket core without the Gemmini, by which the latency is larger than 20 seconds. This shows that the Matrix Multiplication accelerator, i.e. Gemmini, is indispensable.

We also test the workload on three NVIDIA GPUs. To our surprise, the latency on those barely meets that on 4x4 Gemmini plus one Rocket core for small K values. We suppose that the GPU as an off-chip accelerator, has more overhead in communication and synchronization with the CPU than an on-chip accelerator. Moreover, calculations of the small-size matrix in our workload cannot fully demonstrate the advantage of massive parallelism in high-end GPUs. This may also account for why three GPUs of different performance specifications can behave similarly. Nevertheless, the latency by GPU scales in a less serious way with the K value going up.

The third reference is a high-performance x86 CPU with MegaByte-level caches. The latency of our workload on it exceeds that of our architecture of the largest design point. However, if its frequency is scaled down to 500MHz, performance would be comparable, also considering that it is a CPU specialized for intensive computation in the data center.

On top of that, as presented in the last column of table 5.11, we find that the ratio of latency between K-028 and K-001 basically decreases with more cores. This means the latency cost of sequence partition can be relatively refined by parallelism, leading to a cost of 15% to 41% more latency for partition factor 28.

Latency Breakdown

Next, we inquire into how latency components evolve with the system scaling up. As figure 5.9, 5.10 and 5.11 show, the latency of the workload of nine K values changes similarly with increasing parallelism, but not exactly the same, due to the varying proportion of latency components.

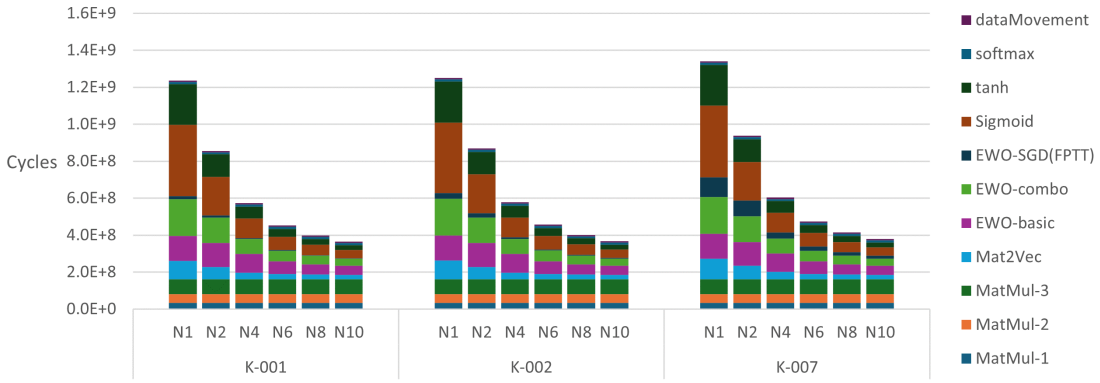

Figure 5.9: Comparison of Cumulative Matrix Multiplication Latency, K∈{1,2,7}, Gemmini 4x4

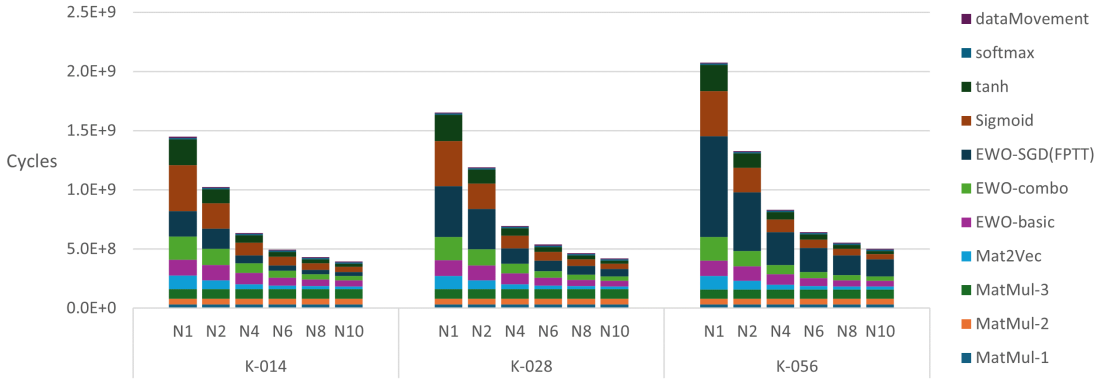

Figure 5.10: Comparison of Cumulative Matrix Multiplication Latency, K∈{14,28,56}, Gemmini 4x4

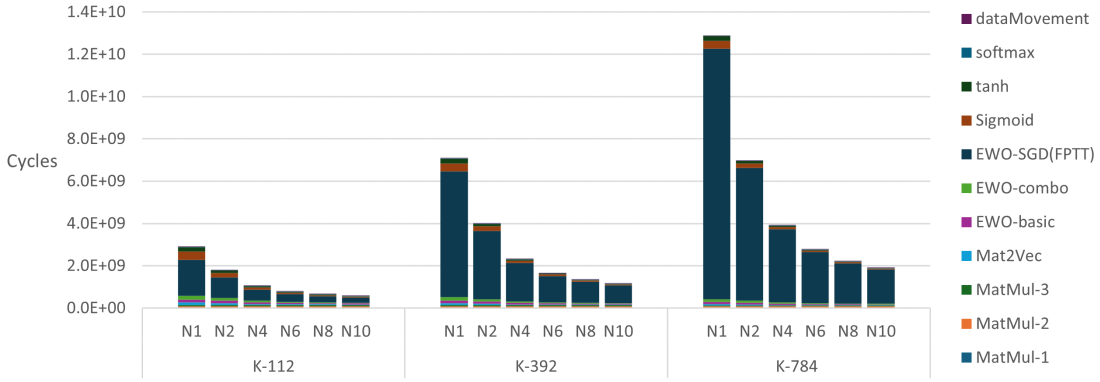

Figure 5.11: Comparison of Cumulative Matrix Multiplication Latency, K∈{112,392,784}, Gemmini 4x4

Since matrix multiplications and element-wise operations (more generally parallelizable operations) are detached as they depend on Gemmini and multicore respectively, we analyze these two types of operations separately.

In figure 5.12, it is noticeable that the larger mesh provides roughly 40% drops on the cumulative matrix multiplication latency in the workload of all K values. Combining statistics in table 5.9, moving on to a larger mesh size can be a barely acceptable trade-off because the resource utilization almost doubles while the latency nearly halves considering the matrix multiplication only. However, as previously shown in table 5.11 a larger mesh cannot make a substantial difference in the total latency for our workload.

On the other hand, taking K-028 as an example, parallelizable functions experience distinct speedups with the system scaling up in figure 5.13. We suppose that the differences result from the computation intensity of the function, i.e., the ratio between the number of computations and the data moving. Functions of lower computation intensity may frequently lead to CPU cores being idle to wait for data from memory, which can hinder the performance gain by parallelism. In figure 5.13, it can be observed that computation-intensive

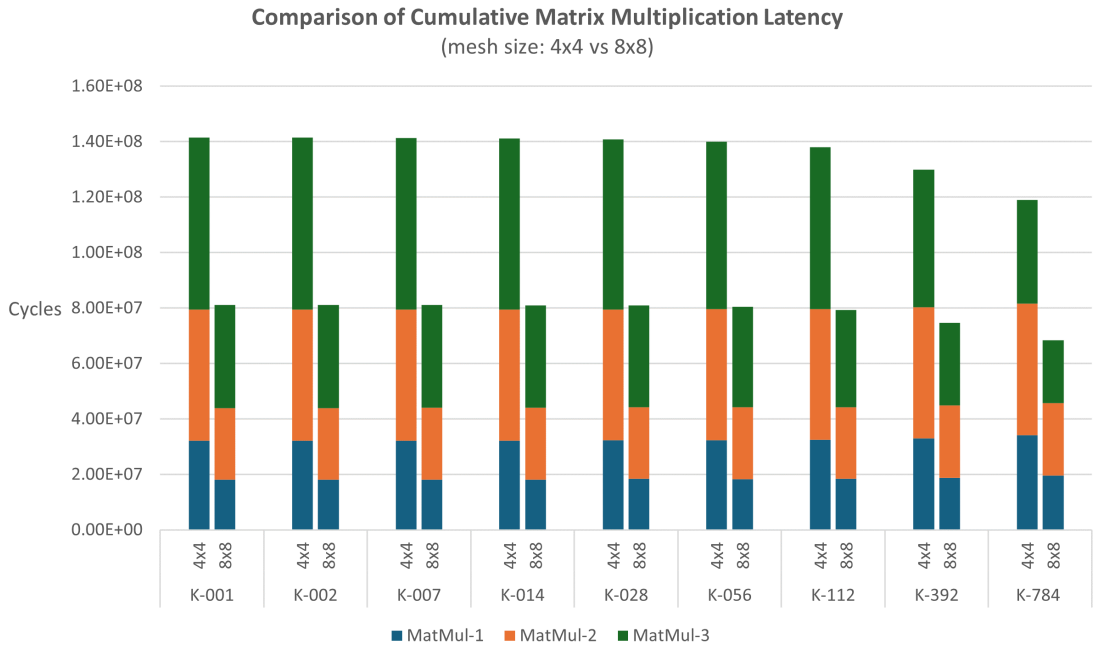

Figure 5.12: Comparison of Cumulative Matrix Multiplication Latency

functions, `EWO-SGD(FPTT)` or `Sigmoid`, have higher speedup than `Mat2Vec` or `EWO-basic`

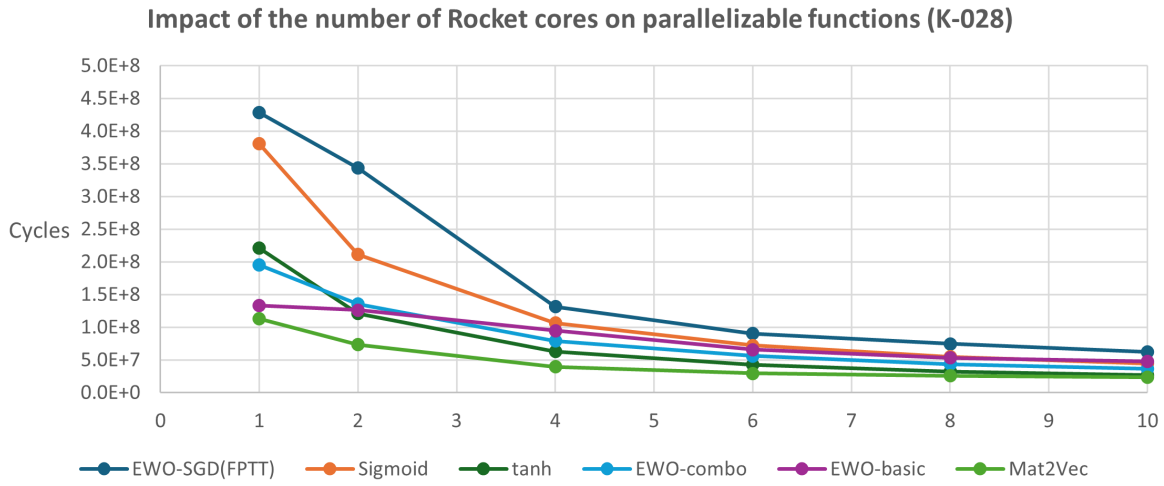

Figure 5.13: Impact of the number of Rocket cores on parallelizable functions

Furthermore, the speedup of the workload depends on the speedup of the sum of parallelizable functions. The accelerations with the workload of nine K values are in figure 5.14. When K increases, the proportion of `EWO-SGD(FPTT)`, which is computation-intensive, goes up in the workload. Therefore, the workload of larger K is accelerated more efficiently on the multicore platform, though all variants cannot achieve the ideal speedup that is identical to the number of cores.

5.3 Discussion

This chapter presents the investigation into Research Question 2. We motivate our choice of Matrix Matrix Multiplication (MMM) accelerator, namely Gemmini, with multiple RISC-V core to accelerate the FPTT featuring MMM, considering on cost, efficiency and flexibility.

Based on RNN's property, we further perform Gemmini compression using the more efficient data type BF16 and reducing unnecessary hardware blocks. Results show a 40% gain in resource utilization and the halved memory footprint, while the trainability is maintained. However, conversion overhead between FP32

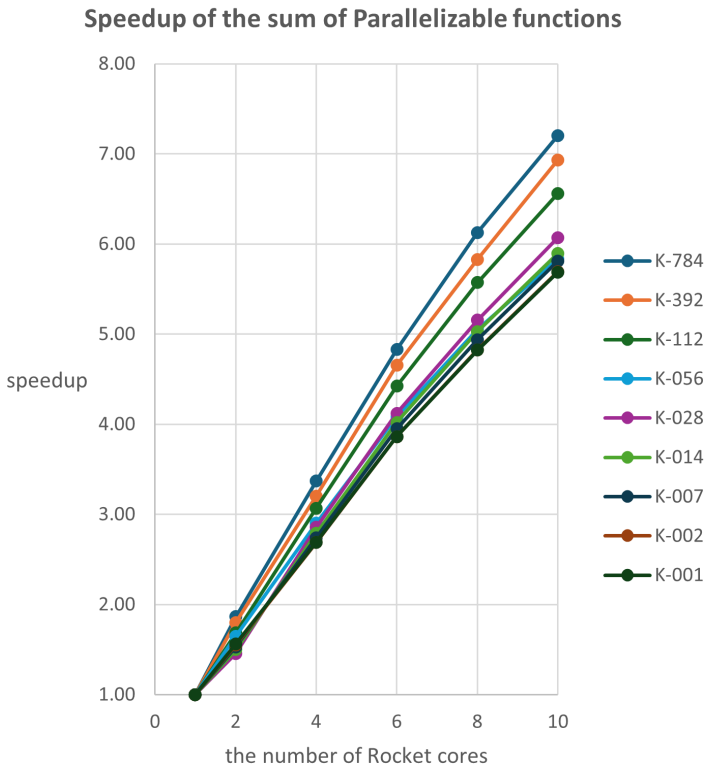

Figure 5.14: Speedup of the sum of parallelizable functions

and BF16 is introduced because of Rocket's limitations.

Moreover, we make the system scalable by tweaking the systolic array size in Gemmini and the number of Rocket cores, leading to a system with up to 8x8 systolic array and 10 Rocket. Results demonstrate that system scaleup can achieve 4 to 7 times speedup on the workload of various computation intensities, at the cost of up to 4.5 times more resource utilization.

Chapter 6

Conclusion

6.1 Contributions

- We derive and implement the deterministic computation flow of FPTT-based training processing for LSTM network by C programming targeting edge computing, and then identify its computation bottleneck in Matrix Matrix Multiplication and element-wise operations.
- To efficiently accelerate the bottleneck in FPTT computations, we propose and optimize a scalable and efficient edge hardware architecture composed of an architecturally compressed 40% smaller but similarly performant BF16 Gemmini (MMM accelerator) and multiple RISC-V Rocket cores, leveraging the Chipyard infrastructure and FPGA-based HW/SW co-design flow.
- We demonstrate the efficacy of our full system solution of hardware and software evolution by training a compact LSTM network with small-batch S-MNIST samples for edge learning. Results show that for the input sequence of length 784, when partition factor K is set to $\sqrt{784} = 28$, sequence partition and datatype BF16 together can bring up 22 times save in memory footprint compared with traditional BPTT training with FP32 while maintaining the trainability, at the cost of 15% to 41% more latency due to scalable parallelism. In general, the scalable architecture can accelerate the training process of various computation intensities (determined by the partition factor) to 4 to 7 times faster compared with the basic case 4x4-1, at the cost of up to 4.5 times more resource utilization.

6.2 Limitations

This study is mainly subject to three limitations.

Matrix Matrix Multiplication (MMM). Unexpectedly, FPTT does not get rid of computationally intensive MMM, unlike some bio-inspired training algorithms for which we may be able to design hardwired functional blocks for extreme performance. Instead, to handle the standalone challenge of MMM, we have to utilize an existing accelerator and turn to focus on a system-level solution. In other words, MMM limits the space of hardware design and makes us carry out the study in an expected direction.

BF16 in RISC-V. To the best of our knowledge, no open-source RISC-V CPU implements BF16, whilst some commercial solutions support that like Greenwaves `GAP9`. The inability of BF16 incurs huge conversion overhead in latency and power.

FPGA-Accelerated Simulation. This work depends on `FireSim`, the FPGA-Accelerated Simulation which has two limitations. First, it is not real FPGA prototyping as peripherals like the SD card and the DDR memory are System C models, to achieve deterministic execution, so the results may deviate from the circumstances with real devices. Then, large designs cannot reach high frequency in the FPGA, i.e. most design points run at 10 MHz for easier placement and routing, which makes it infeasible to test more complicated applications.

6.3 Future Works

Several future works are worth consideration.

BF16 in RISC-V. It may be interesting to develop an open-source solution of BF16 for the RISC-V CPU like Rocket. This process may involve design in the functional model, microarchitecture, compilation, and benchmarks, but it would be generally advantageous for various applications.

Tape-out, real-life applications and a demonstrator. We consider it may be reasonable to make a chip so that the design can run at a frequency, i.e., 500 MHz, feasible for real-life applications, and we can

then develop a live demonstrator to show the effect of model fine-tuning by FPTT. For example, this project[1] implements a routine that the LSTM-based model is first pre-trained on standard GSC, and then fine-tuned on one's individual voice clips. It may be interesting to keep pre-training on GPUs and to switch the fine-tuning to our design.

[1] https://github.com/felixchenfy/Speech-Commands-Classification-by-LSTM-PyTorch

Bibliography

[1] Charlotte Frenkel and Giacomo Indiveri. "ReckOn: A 28nm Sub-mm2 Task-Agnostic Spiking Recurrent Neural Network Processor Enabling On-Chip Learning over Second-Long Timescales". In: *2022 IEEE International Solid-State Circuits Conference (ISSCC)*. Vol. 65. 2022, pp. 1–3. DOI: 10.1109/ISSCC42614.2022.9731734.

[2] Anil Kag and Venkatesh Saligrama. "Training Recurrent Neural Networks via Forward Propagation Through Time". In: *Proceedings of the 38th International Conference on Machine Learning*. Ed. by Marina Meila and Tong Zhang. Vol. 139. Proceedings of Machine Learning Research. PMLR, July 2021, pp. 5189–5200. URL: https://proceedings.mlr.press/v139/kag21a.html.

[3] Everton Gomede. *Exploring Sequential Learning in Artificial Intelligence: Concepts, Applications, and Future Directions*. [Online]. https://medium.com/@evertongomede/exploring-sequential-learning-in-artificial-intelligence-concepts-applications-and-future-d0365573e09f.

[4] Tarun Paparaju. *Exploring Sequential Learning in Artificial Intelligence: Concepts, Applications, and Future Directions*. [Online]. https://medium.com/machine-learning-basics/sequence-modelling-b2cdf244c233.

[5] Sarat Moka Benoit Liquet and Yoni Nazarathy. *The Mathematical Engineering of Deep Learning*. [Online]. https://deeplearningmath.org/.

[6] Ronald J. Williams and Jing Peng. "An Efficient Gradient-Based Algorithm for On-Line Training of Recurrent Network Trajectories". In: *Neural Computation* 2.4 (Dec. 1990), pp. 490–501. ISSN: 0899-7667. DOI: 10.1162/neco.1990.2.4.490. eprint: https://direct.mit.edu/neco/article-pdf/2/4/490/812062/neco.1990.2.4.490.pdf. URL: https://doi.org/10.1162/neco.1990.2.4.490.

[7] Sepp Hochreiter and Jürgen Schmidhuber. "Long Short-Term Memory". In: *Neural Computation* 9.8 (1997), pp. 1735–1780. DOI: 10.1162/neco.1997.9.8.1735.

[8] Wikipedia contributors. *Online machine learning — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=Online_machine_learning&oldid=1195725733. [Online; accessed 22-January-2024]. 2024.

[9] Matthias Delange et al. "A continual learning survey: Defying forgetting in classification tasks". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* (2021), pp. 1–1. ISSN: 1939-3539. DOI: 10.1109/tpami.2021.3057446. URL: http://dx.doi.org/10.1109/TPAMI.2021.3057446.

[10] Brendan McMahan. "Follow-the-Regularized-Leader and Mirror Descent: Equivalence Theorems and L1 Regularization". In: *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*. Ed. by Geoffrey Gordon, David Dunson, and Miroslav Dudík. Vol. 15. Proceedings of Machine Learning Research. Fort Lauderdale, FL, USA: PMLR, Apr. 2011, pp. 525–533. URL: https://proceedings.mlr.press/v15/mcmahan11b.html.

[11] Wikipedia contributors. *IEEE 754 — Wikipedia, The Free Encyclopedia*. [Online; accessed 8-September-2023]. 2023. URL: https://en.wikipedia.org/w/index.php?title=IEEE_754&oldid=1171022504.

[12] Wikipedia contributors. *Unum (number format) — Wikipedia, The Free Encyclopedia*. [Online; accessed 8-September-2023]. 2023. URL: https://en.wikipedia.org/w/index.php?title=Unum_(number_format)&oldid=1169499940.

[13] Hamed Fatemi Langroudi, Zachariah Carmichael, and Dhireesha Kudithipudi. "Deep Learning Training on the Edge with Low-Precision Posits". In: *CoRR* abs/1907.13216 (2019). arXiv: 1907.13216. URL: http://arxiv.org/abs/1907.13216.

[14] Dhiraj D. Kalamkar et al. "A Study of BFLOAT16 for Deep Learning Training". In: *CoRR* abs/1905.12322 (2019). arXiv: 1905.12322. URL: http://arxiv.org/abs/1905.12322.

[15] Alon Amid et al. "Chipyard: Integrated design, simulation, and implementation framework for custom socs". In: *IEEE Micro* 40.4 (2020), pp. 10–21.

[16] Jonathan Bachrach et al. “Chisel: constructing hardware in a scala embedded language”. In: *Proceedings of the 49th Annual Design Automation Conference*. 2012, pp. 1216–1225.

[17] Krste Asanovic et al. “The rocket chip generator”. In: *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17* 4 (2016), pp. 6–2.

[18] Christopher Celio, David A Patterson, and Krste Asanovic. “The berkeley out-of-order machine (boom): An industry-competitive, synthesizable, parameterized risc-v processor”. In: *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2015-167* (2015).

[19] Florian Zaruba and Luca Benini. “The cost of application-class processing: Energy and performance analysis of a Linux-ready 1.7-GHz 64-bit RISC-V core in 22-nm FDSOI technology”. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 27.11 (2019), pp. 2629–2640.

[20] Hyungmin Cho, Jeesoo Lee, and Jaejin Lee. “FARNN: FPGA-GPU hybrid acceleration platform for recurrent neural networks”. In: *IEEE Transactions on Parallel and Distributed Systems* 33.7 (2021), pp. 1725–1738.

[21] Sicheng Li et al. “FPGA Acceleration of Recurrent Neural Network Based Language Model”. In: *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*. 2015, pp. 111–118. DOI: 10.1109/FCCM.2015.50.

[22] Ji Lin et al. “On-Device Training Under 256KB Memory”. In: *Advances in Neural Information Processing Systems*. Ed. by S. Koyejo et al. Vol. 35. Curran Associates, Inc., 2022, pp. 22941–22954. URL: https://proceedings.neurips.cc/paper_files/paper/2022/file/90c56c77c6df45fc8e556a096b7a2b2e-Paper-Conference.pdf.

[23] Haoyu Ren, Darko Anicic, and Thomas A Runkler. “Tinyol: Tinyml with online-learning on microcontrollers”. In: *2021 International Joint Conference on Neural Networks (IJCNN)*. IEEE. 2021, pp. 1–8.

[24] Leonardo Ravaglia et al. “A TinyML Platform for On-Device Continual Learning With Quantized Latent Replays”. In: *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* 11.4 (Dec. 2021), pp. 789–802. ISSN: 2156-3365. DOI: 10.1109/jetcas.2021.3121554. URL: http://dx.doi.org/10.1109/JETCAS.2021.3121554.

[25] Navjot Kukreja et al. “Training on the Edge: The why and the how”. In: *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE. 2019, pp. 899–903.

[26] Geng Yuan et al. “Mest: Accurate and fast memory-economic sparse training framework on the edge”. In: *Advances in Neural Information Processing Systems* 34 (2021), pp. 20838–20850.

[27] A.P. van der Burgt. *AI in the Wild : Robust evaluation and optimized fine-tuning of machine learning algorithms deployed on the edge*. June 2023. URL: http://essay.utwente.nl/95066/.

[28] Bojian Yin, Federico Corradi, and Sander M Bohté. “Accurate online training of dynamical spiking neural networks through Forward Propagation Through Time”. In: *Nature Machine Intelligence* (2023), pp. 1–10.

[29] Charlotte Frenkel et al. “A 0.086-mm^2 12.7-pJ/SOP 64k-Synapse 256-Neuron Online-Learning Digital Spiking Neuromorphic Processor in 28-nm CMOS”. In: *IEEE Transactions on Biomedical Circuits and Systems* 13.1 (2019), pp. 145–158. DOI: 10.1109/TBCAS.2018.2880425.

[30] Hasan Genc et al. “Gemmini: Enabling systematic deep-learning architecture evaluation via full-stack integration”. In: *2021 58th ACM/IEEE Design Automation Conference (DAC)*. IEEE. 2021, pp. 769–774.

[31] Yunsup Lee et al. “The Hwacha vector-fetch architecture manual, version 3.8. 1”. In: *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2015-262* (2015).

[32] Dennis Agyemanh Nana Gookyi et al. “Deep Learning Accelerators’ Configuration Space Exploration Effect on Performance and Resource Utilization: A Gemmini Case Study”. In: *Sensors* 23.5 (2023), p. 2380.

Appendix A

Experiments on Code Implementation of Kag et al.

In this appendix, we present the experiments on code implementation of FPTT algorithm by Kag and Saligrama [2], available on GitHub [1].

The goal is to reproduce the results reported by Kag and Saligrama [2]. Moreover, we find it necessary to investigate the influence of the granularity of online formulation, i.e., K value and the regularization term $R(t)$, since they are directly related to the number of computations and memory accesses when we design hardware acceleration.

A.1 Methodology and Experiment Steup

We use the default data sets and default network size set by Kag and Saligrama [2] as shown in table A.1, and the detailed explanation of data sets can be found at the link [2].

For each data set, we choose a series of K values to inquire into the effect of the granularity of online formulation. We have the K values centering around the square root of sequence length, i.e. $\sqrt{T}$ or the integer close to it, since Kag and Saligrama [2] state that $\sqrt{T}$ leads to best trainability.

Furthermore, for data sets other than `Add Task`, the loss function includes regularization terms $R(t)$, so we test the performance on them with and without $R(t)$ separately.

Table A.1: Data sets used with FPTT

Data Set	LSTM layer(s)	Sequence Length	K values	$R(t)$
PTB-Word-300	256	300	{1, 5, 15, 20, 30, 50, 100, 150, 300}	optional
PTB-Word-70	960x960	70	{1,2,5,7,10,14,35,70}	
PTB-Char	1000x1000x200	150	{1,3,5,10,15,30,50,75,150}	
S-MNIST	128	784	{1,2,7,14,28,56,112,392,784}	
Permuted S-MNIST	128	784	{1,2,7,14,28,56,112,392,784}	
CIFAR-10	128	1024	{1,2,8,16,32,64,128,512,1024}	
Add Task-200	128	200	{1,2,5,10,20,40,100,200}	no
Add Task-1000	128	1000	{1,2,5,10,20,50,100,200,500,1000}	

A.2 Results

Table A.2: Experiment results (Perplexity) of PTB-300

	K=1	K=5	K=15	K=20	K=30	K=50	K=100	K=150	K=300
with reg	**148.42**	**109.50**	**101.52**	**102.61**	**102.35**	**104.37**	**107.24**	112.69	139.21
no reg	164.63	121.22	119.39	104.74	105.53	110.80	108.14	**112.29**	**137.27**

[1] `https://github.com/anilkagak2/FPTT`

[2] `http://proceedings.mlr.press/v139/kag21a/kag21a-supp.pdf`

Table A.3: Experiment results (Perplexity) of PTB-Word-70

	K=1	K=2	K=5	K=7	K=10	K=14	K=35	K=70
with reg	76.08	75.48	74.39	73.91	74.72	75.44	**88.87**	105.70
no reg	**75.20**	**75.44**	**73.75**	**73.80**	**73.56**	**76.38**	90.36	**104.62**

Table A.4: Experiment results (BPC, Bits-Per-Character) of PTB-Char

	K=1	K=3	K=5	K=10	K=15	K=30	K=50	K=75	K=150
with reg	**1.244**	**1.273**	**1.284**	1.300	**1.307**	1.339	1.366	**1.391**	1.488
no reg	1.257	1.280	1.293	1.300	1.308	**1.337**	1.366	1.392	**1.487**

Table A.5: Experiment results (Accuracy %) of SMNIST

	K=1	K=2	K=7	K=14	K=28	K=56	K=112	K=392	K=784
with reg	**61.77**	97.15	**97.44**	97.36	96.47	**96.06**	**55.56**	**27.87**	**18.32**
no reg	11.32	**97.52**	97.22	**97.41**	**96.63**	95.77	16.87	11.92	15.86

Table A.6: Experiment results (Accuracy %) of Permuted SMNIST

	K=1	K=2	K=7	K=14	K=28	K=56	K=112	K=392	K=784
with reg	88.26	**88.22**	**93.99**	94.07	**93.17**	**91.30**	**74.01**	27.70	25.88
no reg	**88.72**	86.50	93.44	**94.37**	93.03	90.09	27.78	**30.19**	**31.72**

Table A.7: Experiment results (Accuracy %) of CIFAR10

	K=1	K=2	K=8	K=16	K=32	K=64	K=128	K=512	K=1024
with reg	49.98	61.59	65.38	63.41	59.82	58.89	**41.11**	**36.93**	**28.32**
no reg	**56.33**	**62.35**	**66.76**	**64.59**	**60.75**	**59.03**	37.96	28.21	22.84

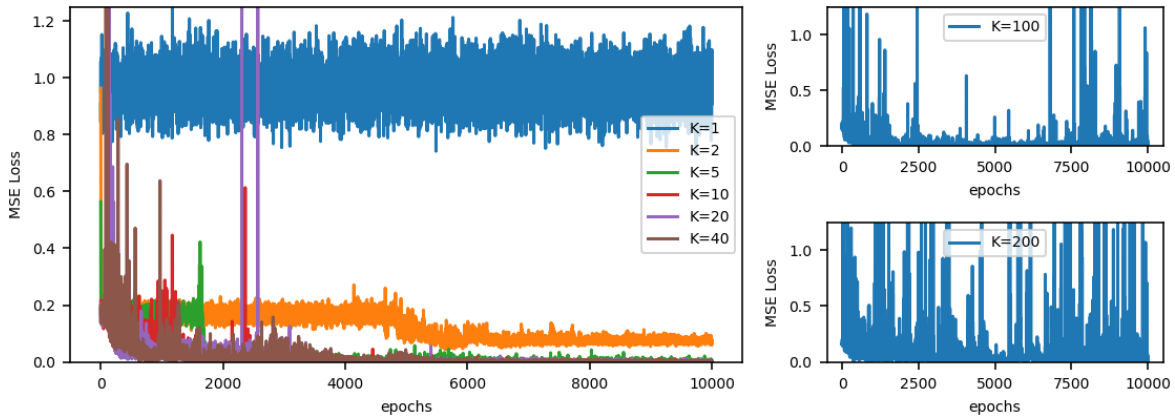

Figure A.1: Experiment results (Mean Squared Error) of Add Task(200)

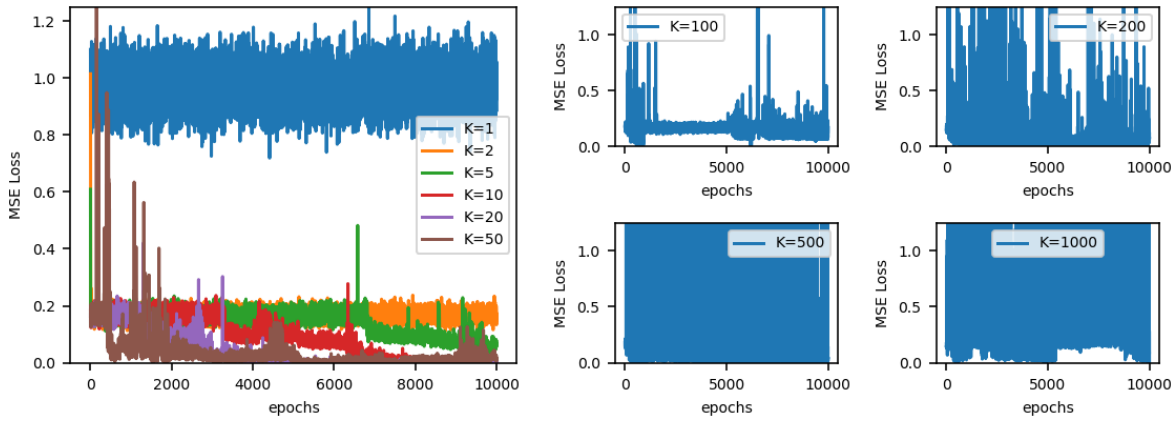

Figure A.2: Experiment results (Mean Squared Error) of Add Task(1000)

Appendix B

Parameters of Gemmini

In section 5.1.4, we perform a reduction on the Gemmnini architecture by changing part of the system parameters to fit our workload efficiently. Here, we list all parameters and divide them into 10 groups depending on the scope with a brief introduction.

Table B.1 lists the datatype to generate in the systolic array for computation. Assuming an expression of matrix multiplication and addition in Exp.B.1. `inputType` is the type of data in input matrices A and B. `spatialArrayOutputType` is for output matrix C, and `accType` defines matrix D that is usually the bias in ANN computation. Also, D serves as an accumulator for the partial sum in the calculation by Gemmini.

$$C = A \cdot B + D \tag{B.1}$$

`Data` is a defined type class that covers `UInt`, `SInt` and `Float`. All these three types have configurable bit width. In `Float(8,24)`, 8 is the width of exponent bits and 24 is the sum of significant bits and the sign bit.

Table B.1: Arithmetics

Parameter	Data Type	Default Value
inputType	`Data`	Float(8,24)
spatialArrayOutputType	`Data`	Float(8,24)
accType	`Data`	Float(8,24)

By parameters in table B.2, we can define the systolic array. `tileRows` × `tileColumns` is the number of Processing Elements (PE) in one tile, while `meshRows` × `meshColumns` determines the number of tiles in the mesh/array.

The `dataflow` can be set to `Dataflow.WS`, `Dataflow.OS` or `Dataflow.BOTH` so that the systolic array can be Weight Stationary, Output Stationary or both.

Besides, tree reduction can be done when there is more than one PE in the tile. We can also configure the latency between tiles and delay the output of the mesh.

Table B.2: Systolic Array

Parameter	Data Type	Default Value
tileRows	`Int`	1
tileColumns	`Int`	1
meshRows	`Int`	4
meshColumns	`Int`	4
dataflow	`Dataflow.Value`	Dataflow.BOTH
use_tree_reduction_if_possible	`Boolean`	true
tile_latency	`Int`	2
mesh_output_delay	`Int`	0
shifter_banks	`Int`	1

Parameters in table B.3 define the entries in the reservation station for `Load`, `Store` and `Execute` commands, which affects the instruction level parallelism in out-of-order execution. In table B.4, two parameters specify the max bytes and the bus width of DMA transfers. On top of that, as shown in table B.5, the TLB size can be configured. We can also choose whether to use the register filter and to share TLB.

Table B.3: Dependency Management

Parameter	Data Type	Default Value
reservation_station_entries_ld	`Int`	8
reservation_station_entries_st	`Int`	4
reservation_station_entries_ex	`Int`	16

Table B.4: DMA Engine

Parameter	Data Type	Default Value
dma_maxbytes	`Int`	64
dma_buswidth	`Int`	128

Table B.5: TLB

Parameter	Data Type	Default Value
tlb_size	`Int`	4
use_tlb_register_filter	`Boolean`	true
use_shared_tlb	`Boolean`	true

In table B.6, three queue lengths determine the length of the instruction queue for Load, Store and Execution commands. In addition, we can add read and write channels between the Execute Controller and Scratchpad/Accumulator.

Table B.6: Controllers: Load, Store and Execution

Parameter	Data Type	Default Value
ld_queue_length	`Int`	8
st_queue_length	`Int`	2
ex_queue_length	`Int`	8
ex_read_from_spad	`Boolean`	true
ex_write_to_spad	`Boolean`	true
ex_read_from_acc	`Boolean`	true
ex_write_to_acc	`Boolean`	true
use_dedicated_tl_port	`Boolean`	false
hardcode_d_to_garbage_addr	`Boolean`	false

Given the parameters in table B.7 and B.8, we can customize the number of banks, ports, capacity, and delay/latency of the scratchpad and the accumulator.

Table B.7: Scratchpad

Parameter	Data Type	Default Value
sp_banks	`Int`	4
sp_singleported	`Boolean`	true
sp_capacity	`GemminiMemCapacity`	CapacityInKilobytes(256)
spad_read_delay	`Int`	1
max_in_flight_mem_reqs	`Int`	16

Parameters in table B.9 are about data scaling. When the data is moved into the systolic array from the scratchpad for calculation or the other way around, it is optional to scale the data by multiplying a value of which the type can be configured. Taking the previous example Exp.B.1, `mvin_scale_args` scales matrices A and B; `mvin_scale_acc_args` scales D; `acc_scale_args` scales C.

By parameters in table B.10, we can also decide whether to generate the hardwired functionality that might accelerate Deep Learning workloads, including activations, normalization, etc. Eight counters are available for profiling the performance of TLB, DMA and reservation stations.

The remaining parameters are given in table B.11. When there are two Gemmini instantiated in the system, `use_shared_ext_mem` determines whether they share the same scratchpad. On top of that, if the user conducts functional simulation by `FireSim`, counters are available. We can specify the clock gating by setting `clock_gate`; `align_to` represent byte alignment.

Table B.8: Accumulator

Parameter	Data Type	Default Value
acc_banks	`Int`	1
acc_singleported	`Boolean`	false
acc_sub_banks	`Int`	-1
acc_capacity	`GemminiMemCapacity`	CapacityInKilobytes(64)
acc_latency	`Int`	2
acc_read_full_width	`Boolean`	true
acc_read_small_width	`Boolean`	true

Table B.9: Data Scaling

Parameter	Data Type	Default Value
mvin_scale_shared	`Boolean`	false
mvin_scale_args	`Option[ScaleArguments[T, U]]`	*see note* [1]
mvin_scale_acc_args	`Option[ScaleArguments[T, U]]`	*see note* [1]
acc_scale_args	`Option[ScaleArguments[T, U]]`	*see note* [1]

[1] `Some(ScaleArguments((t: Float, u: Float) = t * u, 4, Float(8, 24), -1, identity = "1.0", c_str="((x) * (scale ))"))`

Table B.10: Periphery functionality

Parameter	Data Type	Default Value
has_training_convs	`Boolean`	false
has_max_pool	`Boolean`	true
has_nonlinear_activations	`Boolean`	true
has_dw_convs	`Boolean`	true
has_normalizations	`Boolean`	false
has_first_layer_optimizations	`Boolean`	true
num_counter	`Boolean`	8

Table B.11: Others

Parameter	Data Type	Default Value
use_shared_ext_mem	`Boolean`	false
use_firesim_simulation_counters	`Boolean`	false
clock_gate	`Boolean`	false
aligned_to	`Int`	1

Appendix C

Latency Statistics

Functions	4x4-1	4x4-2	4x4-4	4x4-6	4x4-8	4x4-10
MatMul-1	32563347	32660635	32655474	32648590	32639655	32637689
MatMul-2	47515231	47581086	47592645	47595686	47597284	47600357
MatMul-3	80729304	80919321	80883618	80859326	80853385	80822875
Mat2Vec	99794808	66180304	36465033	28490581	25415175	23363732
EWO-basic	135598385	129897413	100817692	69026804	56122005	50187387
EWO-combo	198628702	138105777	81122126	57836900	46774242	38165105
EWO-SGD(FPTT)	15338565	11035373	4539279	3207029	2703105	2258343
Sigmoid	387007682	208872035	106870934	72352234	55234922	44986171
tanh	220615280	122150810	63014088	42814256	32838282	26885336
softmax	11255697	11095974	11107734	11108280	11109353	11109070
dataMovement	7597494	7578825	7642245	7690297	7693587	7722225
total cycles	1236644495	856077553	572710868	453629983	398980995	365738290
total seconds (500MHz)	2.47	1.71	1.15	0.91	0.80	0.73

Table C.1: K=001, Gemmini-4x4

Functions	8x8-1	8x8-2	8x8-4	8x8-6	8x8-8	8x8-10
MatMul-1	18459875	18591159	18601590	18596792	18586582	18602775
MatMul-2	26070988	26084281	26107947	26116393	26110529	26119613
MatMul-3	44276687	44406489	44447244	44427589	44406133	44381520
Mat2Vec	99778092	66043270	36512750	28333490	25202395	23309110
EWO-basic	135483564	129003477	99589710	67579785	56813034	50728073
EWO-combo	198591979	138308480	81794883	58439742	48253532	39235764
EWO-SGD(FPTT)	15336744	11118666	4566334	3171185	2551684	2147343
Sigmoid	381871313	211728879	106889769	72307270	55273388	44967081
tanh	222307808	121677473	63211174	42943060	33068940	27063180
softmax	11116907	11213244	11091168	11090108	11090050	11090215
dataMovement	7561138	7624405	7637481	7685704	7716859	7730106
total cycles	1160855095	785799823	500450050	380691118	329073126	295374780
total seconds (500MHz)	2.32	1.57	1.00	0.76	0.66	0.59

Table C.2: K=001, Gemmini-8x8

Functions	4x4-1	4x4-2	4x4-4	4x4-6	4x4-8	4x4-10
MatMul-1	32566318	32657850	32661580	32654881	32648286	32646678
MatMul-2	47511977	47582281	47584516	47582130	47583627	47593528
MatMul-3	80703624	80841935	80835670	80814565	80806573	80774164
Mat2Vec	101807256	67308841	36996456	28720261	25529197	23455334
EWO-basic	135568321	129830531	100718001	69075588	56026584	50128762
EWO-combo	198519686	137975777	81271214	57711229	46621162	38155426
EWO-SGD(FPTT)	30682249	23573104	9306294	6477659	5484283	4481947
Sigmoid	381819793	208882078	106892543	72373222	55240410	44998382
tanh	222272410	122039412	63155641	42898303	32896298	26922253
softmax	11099535	11108230	11139867	11140718	11139685	11140956
dataMovement	7546087	7575035	7612612	7671943	7672410	7706625
total cycles	1250097256	869375074	578174394	457120499	401648515	368004055
total seconds (500MHz)	2.50	1.74	1.16	0.91	0.80	0.74

Table C.3: K=002, Gemmini-4x4

Functions	8x8-1	8x8-2	8x8-4	8x8-6	8x8-8	8x8-10
MatMul-1	18458549	18601665	18601924	18596693	18588740	18604211
MatMul-2	26066187	26122849	26105348	26114539	26107130	26115614
MatMul-3	44264569	44387012	44402754	44385930	44363372	44340302
Mat2Vec	101754097	67111841	37023536	28574567	25322875	23408516
EWO-basic	135526987	129149935	99152831	67727282	57005816	50625908
EWO-combo	198476117	137961498	81323714	58359807	48153183	39162927
EWO-SGD(FPTT)	30679681	23400910	9265340	6519857	5315410	4606853
Sigmoid	381809456	211770870	106945083	72373755	55308126	45003025
tanh	221872637	121819852	63043250	42830351	32982395	26988450
softmax	11095197	11221160	11141207	11140670	11136873	11141267
dataMovement	7543380	7615228	7630235	7669665	7700195	7716490
total cycles	1177546857	799162820	504635222	384293116	331984115	297713563
total seconds (500MHz)	2.36	1.60	1.01	0.77	0.66	0.60

Table C.4: K=002, Gemmini-8x8

Functions	4x4-1	4x4-2	4x4-4	4x4-6	4x4-8	4x4-10
MatMul-1	32581783	32671162	32673512	32670254	32663008	32660015
MatMul-2	47529918	47574742	47601059	47600830	47601831	47601893
MatMul-3	80505016	80632735	80637226	80612808	80593063	80568447
Mat2Vec	113082814	73452547	39962013	29860356	26229026	24168532
EWO-basic	135019437	129232800	99438988	68298182	55440174	49650348
EWO-combo	197942047	137668382	80645560	57372962	46016415	38004067
EWO-SGD(FPTT)	107374908	85759825	33579315	23156395	19422893	16218725
Sigmoid	386910366	208892011	106880639	72367655	55216379	44984754
tanh	220708436	123394094	62960618	42782007	32780345	26846401
softmax	11227436	11122672	11113814	11111942	11110527	11110585
dataMovement	7584550	7551562	7592273	7639988	7637455	7674905
total cycles	1340466711	937952532	603085017	473473379	414711116	379488672
total seconds (500MHz)	2.68	1.88	1.21	0.95	0.83	0.76

Table C.5: K=007, Gemmini-4x4

Functions	8x8-1	8x8-2	8x8-4	8x8-6	8x8-8	8x8-10
MatMul-1	18474637	18616441	18602978	18602773	18605057	18620808
MatMul-2	26064962	26129470	26129930	26132955	26126798	26126653
MatMul-3	44187715	44334632	44314208	44295334	44279106	44249654
Mat2Vec	113065545	73174325	39979733	29841839	26071806	24058456
EWO-basic	134980414	127664003	97050243	67138756	56290242	50247325
EWO-combo	197886755	137628481	81387412	58226000	47683232	39052874
EWO-SGD(FPTT)	107377334	85361425	32769950	23083054	19393103	16125347
Sigmoid	386895362	208857623	106889135	72341573	55252977	44965211
tanh	220493604	122015734	63030149	42831386	32965210	26977456
softmax	11213963	11092389	11109925	11109326	11108786	11108498
dataMovement	7573839	7546837	7603304	7644988	7669497	7679383
total cycles	1268214130	862421360	528866967	401247984	345445814	309211665
total seconds (500MHz)	2.54	1.72	1.06	0.80	0.69	0.62

Table C.6: K=007, Gemmini-8x8

Functions	4x4-1	4x4-2	4x4-4	4x4-6	4x4-8	4x4-10
MatMul-1	32599455	32694812	32689867	32685876	32679273	32677049
MatMul-2	47538561	47573306	47607295	47607517	47607089	47605399
MatMul-3	80207419	80357361	80338555	80316374	80300316	80276306
Mat2Vec	114187655	74049671	40195755	29908699	26220373	24169807
EWO-basic	134334618	128653137	97753621	67406372	54440082	48832754
EWO-combo	197099869	136774923	80121680	56959605	45077692	37554116
EWO-SGD(FPTT)	214707958	172674993	66542243	46056457	38442649	32629414
Sigmoid	386781583	211912517	106842600	72337244	55173636	44953984
tanh	220650848	121386265	62947075	42764006	32754088	26829752
softmax	11227056	11213021	11112905	11111691	11109957	11109575
dataMovement	7552759	7544101	7535540	7578229	7573993	7607430
total cycles	1446887781	1024834107	633687136	494732070	431379148	394245586
total seconds (500MHz)	2.89	2.05	1.27	0.99	0.86	0.79

Table C.7: K=014, Gemmini-4x4

Functions	8x8-1	8x8-2	8x8-4	8x8-6	8x8-8	8x8-10
MatMul-1	18486108	18617487	18613820	18611873	18607779	18622117
MatMul-2	26075558	26119427	26139499	26144719	26131958	26134088
MatMul-3	44041797	44152294	44169107	44155471	44130667	44114998
Mat2Vec	114162140	73739526	40205919	29894910	26068482	24060986
EWO-basic	134310530	127439287	93385325	66075341	55040297	49313608
EWO-combo	197039715	137056024	81023480	57803329	46873566	38608014
EWO-SGD(FPTT)	214689193	172555192	65796778	45808385	38131649	32100464
Sigmoid	386766059	211613154	106848383	72301074	55221459	44953705
tanh	220432072	121695852	62995618	42811449	32941180	26968295
softmax	11213210	11241127	11107899	11105868	11105939	11106175
dataMovement	7538615	7545155	7540127	7578904	7605824	7614853
total cycles	1374754997	951774525	557825955	422291323	361858800	323597303
total seconds (500MHz)	2.75	1.90	1.12	0.84	0.72	0.65

Table C.8: K=014, Gemmini-8x8

Functions	4x4-1	4x4-2	4x4-4	4x4-6	4x4-8	4x4-10
MatMul-1	32738940	32857391	32852024	32850091	32844858	32841121
MatMul-2	47575239	47644918	47638438	47637285	47641365	47641245
MatMul-3	79621276	79779813	79756025	79722228	79714400	79682756
Mat2Vec	113319901	73486393	39891147	29655125	26030853	23995027
EWO-basic	133171403	126439533	94942196	65898333	53290445	47809277
EWO-combo	195542776	135679931	79491348	56299122	43990734	36957933
EWO-SGD(FPTT)	428871974	343514689	131879193	90790862	74779290	62311198
Sigmoid	381403866	211449466	106773311	72304421	55110837	44919940
tanh	221608319	121302841	62907976	42738132	32723624	26813858
softmax	11094985	11205457	11099822	11097923	11095337	11095067
dataMovement	7504007	7524494	7510970	7543888	7553274	7572792
total cycles	1652452686	1190884926	694742450	536537410	464775017	421640214
total seconds (500MHz)	3.30	2.38	1.39	1.07	0.93	0.84

Table C.9: K=028, Gemmini-4x4

Functions	8x8-1	8x8-2	8x8-4	8x8-6	8x8-8	8x8-10
MatMul-1	18523657	18669765	18654921	18660161	18657757	18674378
MatMul-2	26179527	26208054	26212173	26218027	26209874	26213523
MatMul-3	43712066	43825420	43820255	43809775	43782617	43760459
Mat2Vec	113288599	73151912	39903966	29654774	25862494	23874859
EWO-basic	133125588	124806120	89252207	64713666	53738469	48248722
EWO-combo	195485850	135652831	79337448	57267848	45848347	38124603
EWO-SGD(FPTT)	428899005	343117139	132072757	90727832	74434059	62315898
Sigmoid	381399481	208705100	106780172	72261711	55175909	44914511
tanh	222037311	121878773	62938790	42777489	32892404	26933340
softmax	11139955	11090793	11126949	11126449	11126187	11127658
dataMovement	7507394	7485513	7509487	7542707	7572818	7585032
total cycles	1581298433	1114591420	617609125	464760439	395300935	351772983
total seconds (500MHz)	3.16	2.23	1.24	0.93	0.79	0.70

Table C.10: K=028, Gemmini-8x8

Functions	4x4-1	4x4-2	4x4-4	4x4-6	4x4-8	4x4-10
MatMul-1	32785496	32792849	32810634	32811416	32812234	32818271
MatMul-2	47537812	47567112	47555139	47554372	47550791	47550440
MatMul-3	78470796	78548482	78575380	78574626	78576903	78549212
Mat2Vec	111618814	72420318	39372610	29365824	25729219	23721476
EWO-basic	130965268	120893539	88997212	63663699	51546719	47125434
EWO-combo	198216677	132213663	78049065	54320590	43237321	36772529
EWO-SGD(FPTT)	851421804	494504500	276076079	201198683	166310931	145398729
Sigmoid	380689368	208534364	108079299	73173275	55666853	45390832
tanh	222335748	121712398	62506325	42434604	32462662	26603750
softmax	11074270	11135689	11199742	11199019	11196568	11197499
dataMovement	7444064	7387904	7452623	7505011	7500569	7519617
total cycles	2072560117	1327710818	830674108	641801119	552590770	502647789
total seconds (500MHz)	4.15	2.66	1.66	1.28	1.11	1.01

Table C.11: K=056, Gemmini-4x4

Functions	8x8-1	8x8-2	8x8-4	8x8-6	8x8-8	8x8-10
MatMul-1	18430181	18525262	18518503	18521482	18525851	18548726
MatMul-2	26061904	26109158	26097482	26100925	26106643	26103109
MatMul-3	43039726	43113010	43153544	43149361	43143751	43126822
Mat2Vec	111622085	72011659	39370621	29286454	25459742	23400502
EWO-basic	130973827	121457730	86494165	63407253	52959409	47048620
EWO-combo	198324170	133255496	78685508	55416561	44211969	36671159
EWO-SGD(FPTT)	851500101	494518623	277004944	202053064	167243158	146136667
Sigmoid	380725365	208556008	107999314	73080412	55705435	45368732
tanh	222789168	121759291	62659263	42568422	32719942	26792233
softmax	11091873	11078778	11205475	11204328	11201573	11199014
dataMovement	7430721	7377436	7465523	7494500	7518698	7519513
total cycles	2001989121	1257762451	758654342	572282762	484796171	431915097
total seconds (500MHz)	4.00	2.52	1.52	1.14	0.97	0.86

Table C.12: K=056, Gemmini-8x8

Functions	4x4-1	4x4-2	4x4-4	4x4-6	4x4-8	4x4-10
MatMul-1	32883914	32946078	32971913	32968935	32970279	32966371
MatMul-2	47406947	47510578	47529192	47538687	47536319	47545132
MatMul-3	76059511	76280918	76216549	76185352	76169658	76154921
Mat2Vec	108289552	70130065	38228222	28453779	24904234	22925518
EWO-basic	128848890	119469074	86420502	63016521	50673957	46451984
EWO-combo	187527171	129456547	76480620	53375196	42015395	35882864
EWO-SGD(FPTT)	1705093926	972241491	519820064	357673841	284957867	239603463
Sigmoid	386092120	211439509	107710836	72971372	55624628	45332907
tanh	219782600	121664523	63134846	42896409	32797713	26910841
softmax	11221242	11198101	11063909	11063924	11065185	11066740
dataMovement	7472579	7449632	7443725	7463714	7472248	7492979
total cycles	2910678452	1799786516	1067020378	793607730	666187483	592333720
total seconds (500MHz)	5.82	3.60	2.13	1.59	1.33	1.18

Table C.13: K=112, Gemmini-4x4

Functions	8x8-1	8x8-2	8x8-4	8x8-6	8x8-8	8x8-10
MatMul-1	18780649	18806301	18818312	18818764	18819008	18827466
MatMul-2	26052037	26048850	26057687	26062307	26064055	26070374
MatMul-3	41728280	41866209	41869977	41845753	41816696	41805516
Mat2Vec	108285226	69791309	38242393	28425678	24697758	22931532
EWO-basic	129040034	120564427	85531453	63027162	52411212	47061571
EWO-combo	187638771	130134378	76795222	54949132	44462212	37306706
EWO-SGD(FPTT)	1705590085	973840428	521425705	357109693	286833753	240138069
Sigmoid	386188822	211445558	106801662	72359945	55186679	44957883
tanh	220043293	121609102	63332623	43070870	33059530	27085268
softmax	11209666	11179877	11056756	11056092	11054161	11055969
dataMovement	7487575	7435251	7421508	7449779	7470643	7463443
total cycles	2842044438	1732721690	997353298	724175175	601875707	524703797
total seconds (500MHz)	5.68	3.47	1.99	1.45	1.20	1.05

Table C.14: K=112, Gemmini-8x8

Functions	4x4-1	4x4-2	4x4-4	4x4-6	4x4-8	4x4-10
MatMul-1	33495746	33659325	33673604	33671234	33667697	33674692
MatMul-2	47580234	47642095	47663453	47667987	47674145	47680557
MatMul-3	64418229	64522116	64518861	64503704	64490675	64464944
Mat2Vec	91194591	58955243	32255344	24042104	21019523	19290073
EWO-basic	113346079	102146514	74976036	54400012	44209055	40634785
EWO-combo	162356225	112721683	65638832	45671704	36593759	30582781
EWO-SGD(FPTT)	5948954729	3239488035	1817544256	1247796385	997890292	836132767
Sigmoid	386654900	211240088	106722555	72317953	55070683	44965480
tanh	219910077	121062763	62816558	42713223	32664293	26814139
softmax	11220150	11267846	11115414	11115130	11116317	11115207
dataMovement	8281490	8045814	8011818	8063858	8063418	8110460
total cycles	7087412450	4010751522	2324936731	1651963294	1352459857	1163465885
total seconds (500MHz)	14.17	8.02	4.65	3.30	2.70	2.33

Table C.15: K=392, Gemmini-4x4

Functions	8x8-1	8x8-2	8x8-4	8x8-6	8x8-8	8x8-10
MatMul-1	19247992	19346093	19351040	19351252	19350659	19358140
MatMul-2	26200744	26204975	26217043	26227246	26228357	26240998
MatMul-3	35449656	35571871	35576395	35552464	35536090	35515621
Mat2Vec	91204475	58753610	32254814	23995669	20822154	19054517
EWO-basic	113400066	102151636	72441624	53899691	45501963	40403971
EWO-combo	162485725	112808414	66127850	46992864	36799203	30811875
EWO-SGD(FPTT)	5950333809	3238931235	1816044463	1248529160	998230420	836247778
Sigmoid	386021624	212969790	106759501	72319164	55172572	44978313
tanh	221029656	121371418	62979681	42845683	32926959	27005860
softmax	11320591	11201505	11124422	11123272	11121401	11120251
dataMovement	8277093	8047868	7997566	8044445	8061293	8108910
total cycles	7024971431	3947358415	2256874399	1588880910	1289751071	1098846234
total seconds (500MHz)	14.05	7.89	4.51	3.18	2.58	2.20

Table C.16: K=392, Gemmini-8x8

Functions	4x4-1	4x4-2	4x4-4	4x4-6	4x4-8	4x4-10
MatMul-1	34540715	34626590	34610250	34611225	34605259	34607575
MatMul-2	47872685	47607222	47630205	47632987	47639238	47642502
MatMul-3	48128411	48239010	48188950	48161851	48146237	48121500
Mat2Vec	67186568	43286140	23784826	17682868	15389634	14089602
EWO-basic	93593696	85837753	61248652	44900314	36100197	32753650
EWO-combo	131307735	86988383	50720635	35940382	28359896	24331352
EWO-SGD(FPTT)	11833532639	6281689661	3471557163	2422287964	1909819738	1624406347
Sigmoid	385510441	208330818	106625293	72244293	54996273	44899359
tanh	221001758	121710467	63054598	42876528	32832617	26990026
softmax	11257151	11164461	11115354	11115330	11116288	11117846
dataMovement	9458413	9043634	8986825	9036365	9047043	9131012
total cycles	12883390212	6978524139	3927522751	2786490107	2228052420	1918090771
total seconds (500MHz)	25.77	13.96	7.86	5.57	4.46	3.84

Table C.17: K=784, Gemmini-4x4

Functions	8x8-1	8x8-2	8x8-4	8x8-6	8x8-8	8x8-10
MatMul-1	19774716	19877418	19887451	19896844	19894391	19909068
MatMul-2	26437220	26208384	26183713	26195061	26205616	26221891
MatMul-3	26817453	26839199	26804991	26777970	26760427	26744156
Mat2Vec	67191848	43201372	23805896	17674415	15247773	14040307
EWO-basic	93685457	85328727	60238107	44392120	38255570	33113769
EWO-combo	131535674	88050396	51372488	37135759	28581952	25339919
EWO-SGD(FPTT)	11836523704	6282114568	3471878192	2420929783	1919739123	1622010416
Sigmoid	385738895	208226297	106621371	72275886	55101681	44908546
tanh	220979275	121644347	62934341	42836838	32936799	27047431
softmax	11338191	11165233	11088059	11091042	11092059	11092918
dataMovement	9497218	9066236	9006678	9051945	9034141	9135661
total cycles	12829519651	6921722177	3869821287	2728257663	2182849532	1859564082
total seconds (500MHz)	25.66	13.84	7.74	5.46	4.37	3.72

Table C.18: K=784, Gemmini-8x8

Appendix D

Default BF16 Config Debugging

The provided default BF16 configurations of Gemmini in ConfigsFP.scala (line 98) lead to computation error.

The problem results from **Silent Truncation**, a typical error-prone feature with `Chisel` and its toolset, so we have it recorded in this appendix. We debug the design by RTL simulation with the waveform and Chisel source code inspection.

For file PE.scala (line 23) or figure D.1, the compiler would not check or warn the bit-width mismatch between both sides of the assignment. Therefore, designers need to ensure the bit-width matching by manually setting `cType` and `dType` to be identical.

```
14  class MacUnit[T <: Data](inputType: T, cType: T, dType: T) (implicit ev: Arithmetic[T]) extends Module {
15    import ev._
16    val io = IO(new Bundle {
17      val in_a  = Input(inputType)
18      val in_b  = Input(inputType)
19      val in_c  = Input(cType)
20      val out_d = Output(dType)
21    })
22
23    io.out_d := io.in_c.mac(io.in_a, io.in_b)
24  }
```

Figure D.1: PE module of Gemmini

Appendix E

FireSim: FPGA-Accelerated Simulation Bench

The webpage[1] archives the document of `FireSim`. In short, three "machines" constitute the `FireSim` framework, and they can be mapped to the same computer platform or three different ones. Note that all three machines are expected to run the Linux operating system.

`Build Machine` is the one to build the bitstream of the design by Vivado. `Run Machine` is a desktop attached with a `FireSim` compatible FPGA[2] via PCI-E slot. `Manager Machine` remotely accesses and manipulates two other machines through SSH.

We can inspect and interact with the execution of the design on the FPAG by `screen` sessions.

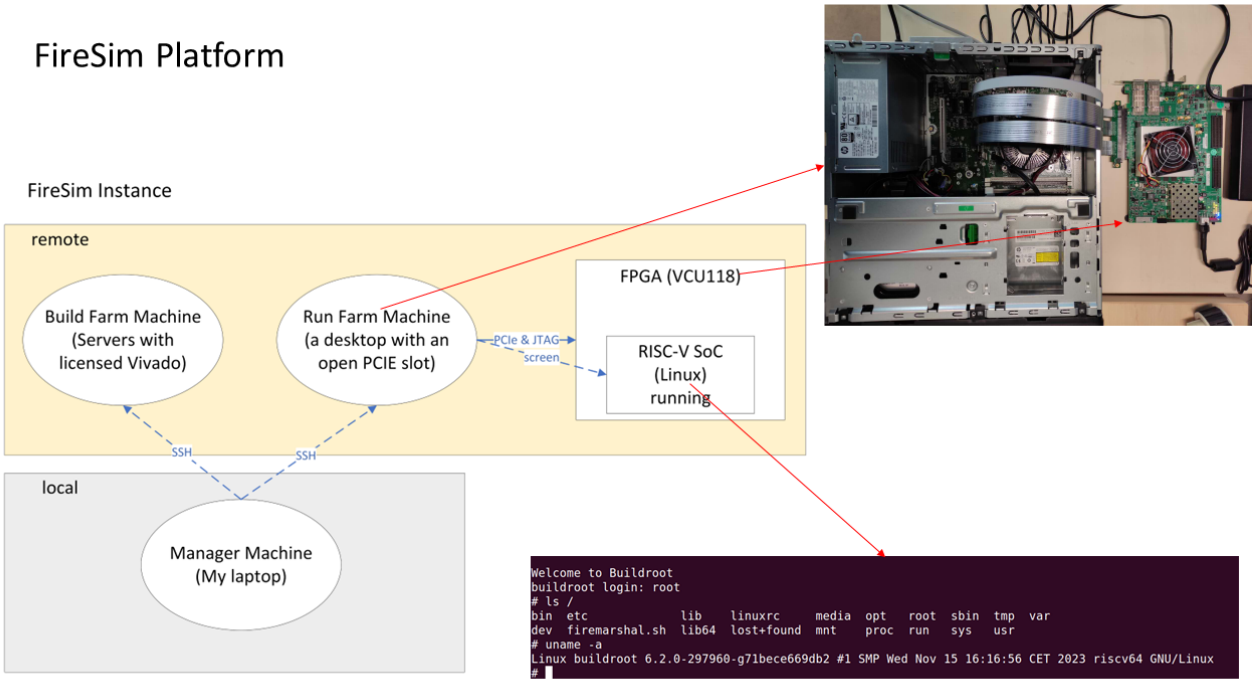

Figure E.1: FireSim

[1] `https://docs.fires.im/en/stable/index.html`

[2] we use AMD Virtex UltraScale+ FPGA VCU118 Evaluation Kit