Imperial College of Science, Technology and Medicine
Department of Computing

Reconfigurable Acceleration of Recurrent Neural Networks

Zhiqiang Que

Submitted in part fulfilment of the requirements for the degree of
Doctor of Philosophy in Computing of the University of London and
the Diploma of Imperial College, August 2023

Abstract

Recurrent Neural Networks (RNNs) have been successful in a wide range of applications involving temporal sequences such as natural language processing, speech recognition and video analysis. However, RNNs often require a significant amount of memory and computational resources. In addition, the recurrent nature and data dependencies in RNN computations can lead to system stall, resulting in low throughput and high latency. This work describes novel parallel hardware architectures for accelerating RNN inference using Field-Programmable Gate Array (FPGA) technology, which considers the data dependencies and high computational costs of RNNs.

The first contribution of this thesis is a latency-hiding architecture that utilizes column-wise matrix-vector multiplication instead of the conventional row-wise operation to eliminate data dependencies and improve the throughput of RNN inference designs. This architecture is further enhanced by a configurable checkerboard tiling strategy which allows large dimensions of weight matrices, while supporting element-based parallelism and vector-based parallelism. The presented reconfigurable RNN designs show significant speedup over CPU, GPU, and other FPGA designs.

The second contribution of this thesis is a weight reuse approach for large RNN models with weights stored in off-chip memory, running with a batch size of one. A novel blocking-batching strategy is proposed to optimize the throughput of large RNN designs on FPGAs by reusing the RNN weights. Performance analysis is also introduced to enable FPGA designs to achieve the best trade-off between area, power consumption and performance. Promising power efficiency improvement has been achieved in addition to speeding up over CPU and GPU designs.

The third contribution of this thesis is a low latency design for RNNs based on a partially-folded hardware architecture. It also introduces a technique that balances initiation interval of multi-layer RNN inferences to increase hardware efficiency and throughput while reducing latency. The approach is evaluated on a variety of applications, including gravitational wave detection and Bayesian RNN-based ECG anomaly detection. To facilitate the use of this approach, we open source an RNN template which enables the generation of low-latency FPGA designs with efficient resource utilization using high-level synthesis tools.

Statement of Originality

I, Zhiqiang Que of Imperial College London, hereby confirm that this thesis is a presentation of my original research work. It does not contain material that has been submitted for a degree in any other institution. When the contributions of others are involved, every effort is made to indicate this clearly in the references to the literature and acknowledgment of collaborative research.

Copyright Declaration

Acknowledgements

First of all, I would like to express my sincere gratitude to my advisor, Professor Wayne Luk. I have learnt a lot from his extensive knowledge in the field of reconfigurable computing and design automation, as well as his passion for research and his kindness in supporting young researchers. I am grateful for his continuous support, encouragement and immense amount of patience during my Ph.D. studies.

I would also like to thank my master project advisor, Professor Yongxin Zhu, for introducing me to the field of computer architecture and reconfigurable computing research and inspiring me throughout these years.

I am grateful to have the opportunity to study in the Custom Computing Group in the Department of Computing at Imperial College London. I would like to express my gratitude to all the colleagues I worked with at Imperial, especially to the members of the Custom Computing group, with whom I collaborated closely for the invaluable discussions and collaboration.

I would also like to express special thanks to the collaborators from the University of British Columbia, the Tokyo Institute of Technology, Corerain Technologies, Intel, Xilinx/AMD and CERN for their support and collaborations on our projects. I am deeply appreciative of the support and contributions they provided, and I feel honored to have worked with such talented and dedicated individuals.

Finally, I would like to give my warmest thanks to my family for their continued support throughout the years.

Contents

List of Tables

List of Figures

Glossary

AI Artificial Intelligence

ANN Artificial Neural Network

ARIMA AutoRegressive Integrated Moving Average

ASIC Application-Specific Integrated Circuit

AUC Area Under the receiver operating characteristic Curve

BB-FIFO Blocking-Batching First In First Out Buffer

BiLSTM Bidirectional Long-Short Term Memory

BPTT Backpropagation Through Time

BRAM Block Random Access Memory

CAT Configurable Adder-tree Tail

CGRA Coarse-Grained Reconfigurable Architecture

CNN Convolutional Neural Network

CPU Central Processing Unit

De-Quant De-Quantization

DMA Direct Memory Access

DNN Deep Neural Network

DRAM Dynamic Random-Access Memory

DSP Digital Signal Processor

EP Element-based Parallelism

FPGA Field-Programmable Gate Array

FPR False Positive Rate

GNN Graph Neural Network

GOPS Giga Operations Per Second

GPU Graphics Processing Unit

GRU Gated Recurrent Unit

GW Gravitational Wave

HLS High-Level Synthesis

HW Hardware

II Initiation Interval

LHC Large Hadron Collider

LIGO Laser Interferometer Gravitational-Wave Observatory

LRCN Long-term Recurrent Convolutional Network

LSTM Long-Short Term Memory

MAC Multiply–Accumulate

MVM Matrix-Vector Multiplication

MUX Multiplexer

NLP Natural Language Processing

NPE Number of Processing Elements

NPU Neural Processor Unit

OCR Optical Character Recognition

PE Processing Element

RENOWN REcurrent Neural networks with finegrained cOlumn-Wise matrix-vector multiplicatioN

RNN Recurrent Neural Network

ROC Receiver Operating Characteristic

RTL Register-Transfer Level

SBE Stall-free Blocking-batching Engine

SIMD Single Instruction Multiple Data

TNN Transformer Neural Network

TOPS Tera Operations Per Second

TPR True Positive Rate

TPU Tensor Processing Unit

TS Timestep

TSP Tensor Streaming Processor

VP Vector-based Parallelism

Chapter 1

Introduction

1.1 Challenges and Objectives

Artificial neural networks (ANNs) are a popular type of machine learning algorithm and form the basis of deep learning. They are computing systems inspired by the structure and function of the human brain, consisting of interconnected nodes that mimic the way biological neurons communicate with each other.

Recurrent neural networks (RNNs) are a type of ANN that have feedback connections, allowing output from some nodes to be fed back as input. This structure allows them to retain information about the sequence of input data. Before the development of RNNs, ANNs were unable to consider the correlation between current data and previous data, as each datum was treated as independent. However, many tasks require current and previous data values when making decisions. RNNs are well-suited for these situations as they are able to incorporate previous information. RNNs have been successful in a variety of sequence-to-sequence processing applications, such as language translation [4, 5, 6], speech recognition [7, 8, 9] and video analysis [10, 11]. Data [12] from Google's datacenters indicate that RNNs accounted for 29% and 21% of Tensor Processing Unit (TPU) workloads in 2016 and 2019, respectively, showing

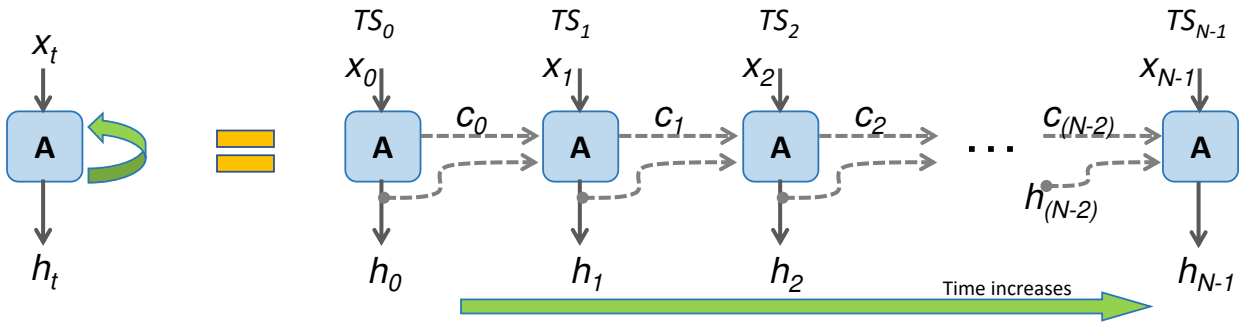


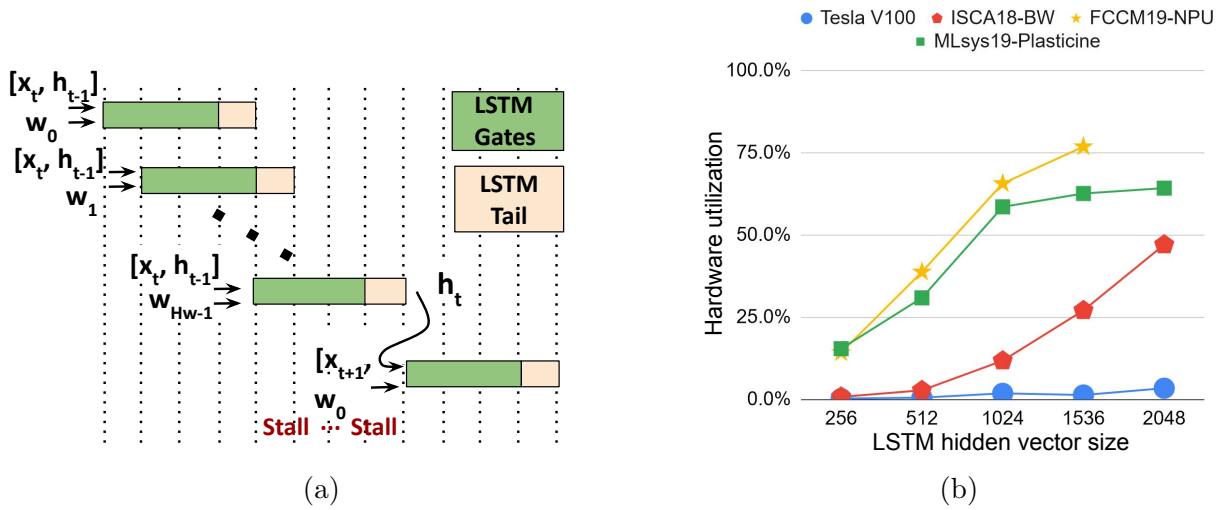
Figure 1.1: A single-layer RNN network (left) showing inter-timestep data dependencies after unfolding (right). **A** represents a hardware unit for the layer. The number of total timesteps is N. TS_i denotes the timestep i, and x_i is the input vector at timestep i. c_i and h_i denote the cell state and the hidden vector at timestep i respectively.

their significance. The most widely-used type of RNN is the long short-term memory (LSTM) network [13], which forms the foundation for many of the aforementioned applications. More about LSTM and other types of RNNs can be found in Section 2.1.2.

Since low latency is crucial for providing a smooth user experience in applications like Apple Siri and Google Voice Search, efficient and real-time acceleration of RNNs is necessary. However, despite their widespread use, hardware acceleration of RNNs is challenging due to the data dependencies arising from their recurrent nature. In an RNN model, a layer or timestep should wait until its preceding layer or timestep is finished, because part of its inputs comes from the output of the preceding layer or timestep. For example, as shown in Fig. 1.1, the computation that requires c_0 and h_0 at timestep TS_1 cannot start until both c_0 and h_0 are available from the preceding timestep. More about data dependencies of RNNs can be found in Section 2.1.3.

1.1.1 First Challenge and Objective

The goal of this thesis is to optimize reconfigurable accelerators for RNNs, with a focus on Field-Programmable Gate Arrays (FPGAs), in order to improve the performance and efficiency of RNN designs on a single FPGA. FPGAs are integrated circuits that have a matrix of logic elements and interconnections that can both be reconfigured to perform a given digital function. By creating multiple copies of these functions, FPGAs are especially adept at implementing parallel functions, making them highly suitable as hardware accelerators for applications with



(a) (b)

Figure 1.2: (a) Pipeline analysis of LSTMs with Row-wise MVM. x_t and h_t denote the input vector and the hidden vector at timestep t, respectively. Details of the pipeline analysis can be found in Section 3.2 with results in Section 3.5. The details of LSTMs and the Gates/Tails can be found in Section 2.1.2. (b) The hardware utilization of various designs. The Brainwave, shown as ISCA18-BW, is labeled in red.

high levels of parallelism. The details of FPGAs are discussed in Section 2.1.5.

However, there are several challenges that hinder the performance of RNNs on FPGAs. The first challenge (C1) is as follows:

- C1: The data dependencies arising from the recurrent nature of RNN computation can lead to undesired system stalls. In addition, inefficient tiling strategies for tiled matrix multiplications can leave hardware resources idle, resulting in low hardware utilization (the proportion of hardware doing useful computation).

Objective O1 of this research is to address C1. The related contribution is summarized below; more information can be found in Section 1.2.1.

The data dependencies in RNNs result in undesired system stall [1, 2] until the required hidden vectors return from the full pipeline to start the next time-step calculation, as shown in Fig. 1.2a. Moreover, inefficient tiling strategies for matrix multiplications can result in hardware resources idle [1, 2]. For example, the hardware utilization of Brainwave [1] ranges from less than 1% to only about 50% for various LSTM models, as shown in Fig. 1.2b. With the need for high performance systems, it is essential to maximize hardware utilization to achieve the highest

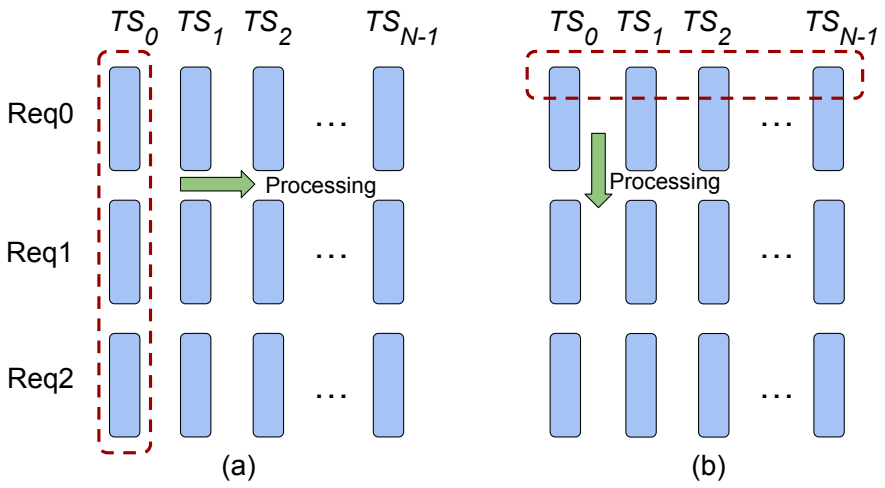


Figure 1.3: (a) The conventional batching of 3 single-layer RNN inference requests. (b) The proposed batching-blocking approach targets on a single request and the requests are processed one by one.

possible effective performance and energy efficiency. To address these issues, Chapter 3 presents several hardware optimizations, including column-wise Matrix-Vector Multiplication (MVM) and a checkerboard tiling strategy, to increase the hardware utilization of RNNs on FPGAs.

1.1.2 Second Challenge and Objective

While Chapter 3 focuses on cloud-based persistent RNNs [1, 2, 14, 15, 16] with weights on-chip, Chapter 4 targets large RNNs such that their weights cannot be stored on-chip, such as those targeting small chips used in edge computing. The second challenge (C2) is as follows:

- C2: In RNN inferences, weights fetched from off-chip memory are typically used only once in computation in each timestep, which is inefficient. Additionally, it is difficult to exploit the parallelism between timesteps in a single inference due to the data dependencies in RNNs.

Objective O2 of this research is to address C2. The related contribution is summarized below; more information can be found in Section 1.2.2.

When an RNN model is so large that the weights have to be stored in off-chip memory, it is not efficient since the fetched weights are typically used only once for each output computation.

The situation is even worse when considering a small embedded system that has small on-chip memory and low memory bandwidth but requires low power consumption. To address this challenge, many previous studies have proposed batching RNN inference requests [17, 18, 19, 20, 21, 22] on FPGAs, as shown in Fig. 1.3 (a). After combining the input vectors from various requests for the same model, the weights can be reused among them, resulting in a good communication-to-computation ratio. Besides, there is no data dependency between different RNN requests. With batches of identically-sized samples, Graphics Processing Units (GPUs) usually have better performance but higher power consumption than FPGAs [1, 19] since GPU kernels have a higher clock frequency. However, this batch technique can harm latency because different requests may not arrive at the same time [20], which means that a newly arrived request must wait until the batch is formed, bringing a latency penalty. In addition, input requests may be based on different neural architectures, which also breaks the assumption that one can conduct computation on batches of identically-sized samples [23]. Thus, the new hardware architecture which can function efficiently with a batch size of one (i.e., no batch) is demanding [23].

To address these issues, Chapter 4 describes a novel approach to increase the performance of RNNs while still targeting a single inference request, by exploiting the parallelism between various timesteps, as shown in Fig. 1.3 (b). By reusing the weights of various timesteps and eliminating the need to wait for other requests as required in general batching techniques, Chapter 4 offers a more efficient approach to address these issues. However, this is challenging because there are data dependencies between different timesteps of RNNs. To overcome this issue, we propose a novel blocking-batching strategy to optimize the throughput of FPGA-based RNN designs by reusing the RNN weights. Our approach includes a performance analysis that can be used to balance trade-offs between area, power and performance (Section 4.2.3).

1.1.3 Third Challenge and Objective

Both Chapters 3 and 4 are based on a fully-folded hardware architecture, as shown in Fig. 1.4(a). It employs a single computational engine with multiple identical Processing Elements (PEs) to

process a tile of the matrix computation of an RNN timestep or an entire timestep of a layer at a time, such that the entire RNN is processed by repeatedly running the engine [1, 2]. However, when the targeted RNN layers are small, these PEs may not be fully utilized, resulting in low hardware efficiency. If an architecture is designed for a certain matrix tile size $N \times M$, when dealing with matrices with smaller tile size, e.g., $n \times m$ such that $n < N$ and $m < M$, then padding would be needed. For example, the engine in Brainwave [1] has a total of 96,000 PEs and can effectively process a 400×240 matrix tile in parallel. Any small RNNs with MVMs smaller than these dimensions that the engines are designed to process will leave some resources idle on these designs, resulting in hardware underutilization and low efficiency. The larger the gap, the greater the decline in actual performance compared to peak performance. In addition, since there is only one single engine in fully-folded architectures, the various layers of a network must have the same amount of parallelism which is not flexible and does not take full advantage of the customizability of FPGAs.

The third challenge (C3) is as follows:

- C3: Existing fully-folded hardware architectures are inefficient for small-sized multi-layer RNNs because they leave some hardware resources idle when the MVMs in the RNNs are smaller than the dimensions that the engines are designed to process [1, 2, 24]. The single engine requires uniform parallelism across layers, reducing flexibility and limiting customization in FPGAs. Moreover, unbalanced Initiation Intervals (IIs) in multi-layer RNN designs result in long latency and low hardware efficiency. Furthermore, there is a lack of publicly available low latency High-Level Synthesis (HLS) based RNN templates.

Objective O3 of this research is to address C3. The related contribution is summarized below; more information can be found in Section 1.2.3.

Chapter 5 proposes a partially-folded hardware architecture that addresses the limitations of fully-folded architectures for processing RNNs. The architecture is shown in Fig. 1.4(b). In this architecture, each layer or a few cascaded layers are implemented using a separate hardware engine, and these engines are connected together to form a coarse-grained pipeline. The main

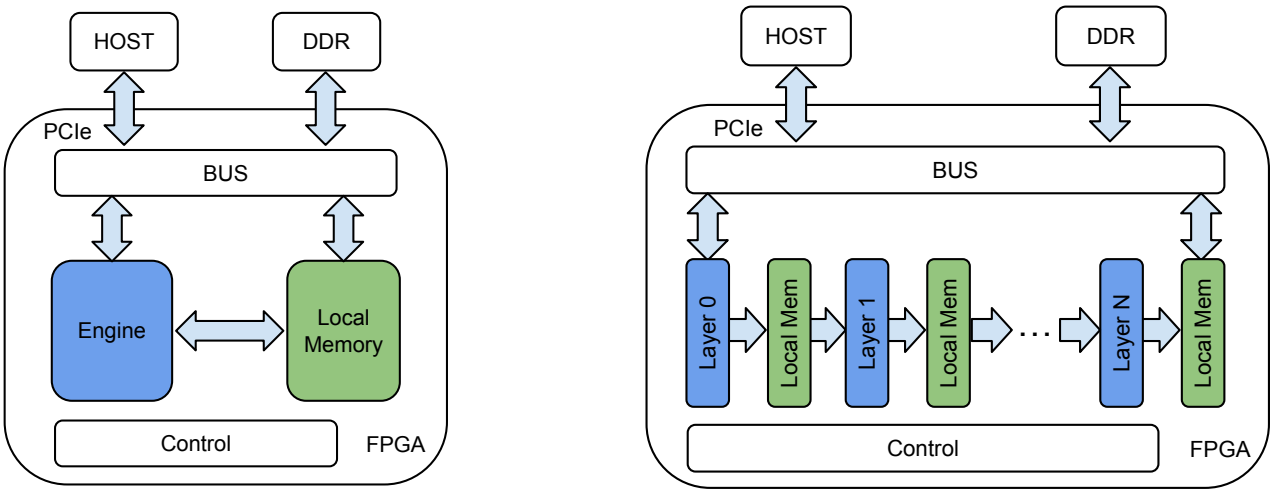


Figure 1.4: (a) A fully-folded hardware architecture with a single computation engine capable of processing multiple layers in an RNN model. (b) A partially-folded hardware architecture with several custom engines, each processing a few timesteps of an RNN layer or a whole layer or even multiple layers.

idea is that if a single engine with a fully-folded architecture is underutilized when running a multi-layer model because of its size, it would be more efficient to split this engine into several smaller partially-folded engines that can run simultaneously, each of which is dedicated to processing a layer or a few cascaded layers with independent and tailor-made optimization. This increases the hardware utilization and reduces the design latency. Moreover, this architecture employs an II balancing technique to balance the IIs among multiple engines, further improving hardware efficiency.

For example, consider the 3-layer RNN network in Fig. 1.5(a) which illustrates inter-timestep and inter-layer data dependencies. There are opportunities to fold the computations along the timestep axis and the layer axis. Fig. 1.5(b) corresponds to fully folding the computations along both axes, resulting in a single engine design, while Fig. 1.5(c) corresponds to partially folding the computations along the timestep axis, resulting in a multi-engine design with the same number of engines as the number of layers.

In addition, the proposed partially-folded architecture aims to achieve low latency and high throughput by fully flattening computations within one timestep for each layer and implementing each operation in the timestep physically on-chip using dedicated hardware circuits.

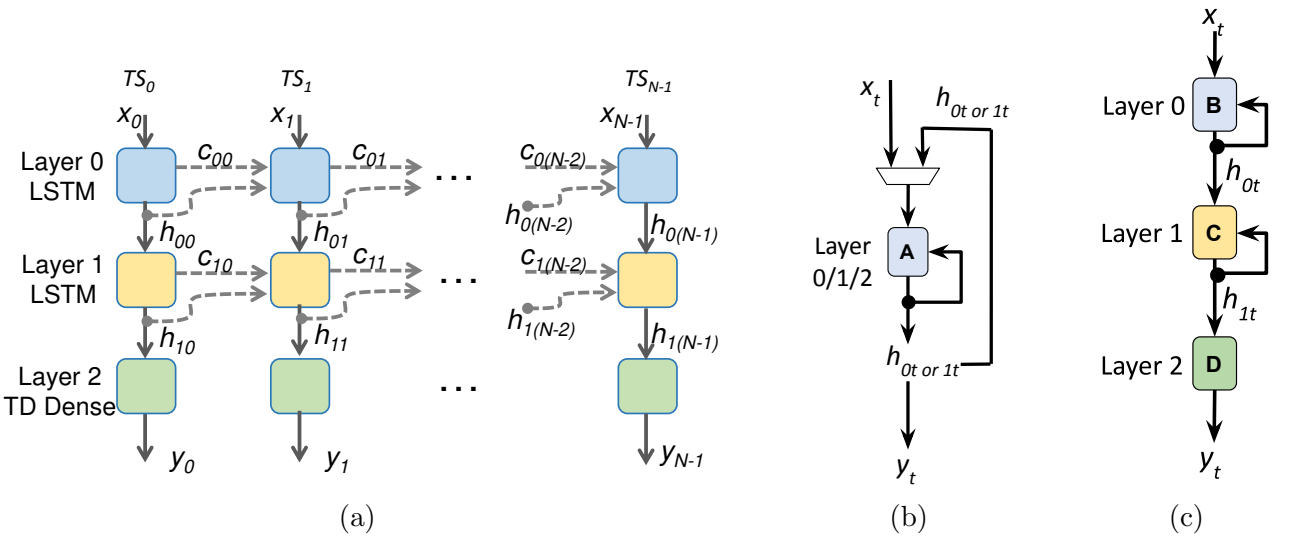


Figure 1.5: (a) A 3-layer RNN network showing inter-layer and inter-timestep data dependencies after unfolding. It has two LSTM layers and one TimeDistributed (TD) dense layer. (b) The design based on the fully-folded architecture. There is only one generic reusable hardware unit, **A**, that processes all the layers repeatedly. The inner loop processes different timesteps of layer 0 and 1, while the outer loop processes different layers. (c) The design based on the partially-folded architecture. There are 3 cascaded layer-dedicated hardware units, **B, C, D**, each processing one layer.

To achieve the best performance, one could fully unfold the computations both along the timestep axis and the layer axis. In this scenario, many engines are connected in a mesh-like 2-dimensional structure like Fig. 1.5(a). Each engine would perform the computation of one timestep, and the entire design could run in a coarse-grained pipeline with an initiation interval as the pipeline depth of one engine. However, this fully-unfolded architecture may not be practical in practice due to limited hardware resources available in current FPGAs.

The design presented in Chapter 5 only unfolds computations along the layer axis, but does not physically unfold the timesteps. Instead, it processes each timestep in the same layer repeatedly using the same hardware unit, taking advantage of the fact that the computation patterns of different timesteps in the same layer are the same. This allows the design to reuse hardware resources efficiently while still maintaining good performance.

Alternatively, when the number of timesteps is small but the number of layers is large, one may unfold computations along the timestep axis, resulting in a multi-engine design with the same number of engines as timesteps rather than layers. However, this approach has the same

drawback as fully-folded architectures, in that the engine must be able to process various layers, reducing the possibility of simplifying each engine to improve efficiency. It is the same reason that the partially-folded architecture designs in Chapter 5 do not use multiple copies of the engine proposed in Chapter 3 and Chapter 4. In addition, unfolding the computations along the timestep axis may require a significant amount of hardware resources since the number of timesteps is typically much larger than the number of layers [25]. For example, a 4-layer LSTM model [26] for gravitational wave detection has 100 timesteps, and a 5-layer LSTM model [27] for speech recognition may have 1500 timesteps.

There are many other ways of folding, such as partially folding computations along the timestep or layer axis, resulting in a mapping of an NL-layer TS-timestep model to an $M \times P$-engine architecture, where $M < NL$ and $P < TS$. Moreover, there are also many ways to interconnect the multiple engines other than the chain-like structure discussed in this thesis, such as central-bus or crossbar-based topologies [28]. However, addressing these topics is beyond the scope of this thesis and is left for future research. Furthermore, this thesis focuses on exploring solutions on a single FPGA and exploring the use of multiple FPGAs is left for future research.

1.2 Research Contributions

This thesis focuses on optimizing reconfigurable accelerators for RNNs on FPGAs. The aim is to improve the performance and efficiency of RNN designs on these platforms. However, there are many challenges that hinder the performance of RNNs on FPGAs, and three main challenges are described in Section 1.1. To address these challenges, this thesis presents three main contributions: a latency-hiding architecture that utilizes column-wise matrix-vector multiplication with a flexible checkerboard tiling strategy, a blocking-batching strategy to reuse RNN weights to optimize the throughput of large RNNs that cannot fit into on-chip memory, and a low latency RNN design for FPGAs based on a partially-folded architecture. These contributions are described in more detail below.

1.2.1 Column-wise Matrix-Vector Multiplication for RNNs

The first contribution of this thesis is a latency-hiding architecture that utilizes column-wise matrix-vector multiplication with a flexible checkerboard tiling strategy to address Challenge C1 and Objective O1.

This contribution (Chapter 3) presents a novel column-wise MVM for RNNs to eliminate data dependencies and introduces a latency-hiding hardware architecture with hybrid kernels as well as Configurable Adder-tree Tail (CAT) units, increasing the hardware utilization and design throughput. It also introduces a flexible checkerboard tiling strategy that supports Element-based Parallelism (EP) and Vector-based Parallelism (VP) to exploit the available parallelism while increasing hardware utilization. In addition, the (EP, VP) parameter space is comprehensively explored. The proposed approach and optimizations are applied to RNN workloads from the DeepBench suite [27]. Compared to Brainwave design [1] and the Brainwave-like NPU [2] with the same RNN workloads on FPGAs, our design achieves 3.7 to 14.8 times better performance and has the highest hardware utilization. This work has been published in papers [24, 29].

1.2.2 Optimizing Large RNNs with Weights Reuse

The second contribution of this thesis is a blocking-batching strategy that reuses the RNN weights to optimize the throughput of large RNNs that are too large to fit into on-chip memory on FPGAs, addressing Challenge C2 and Objective O2.

When RNN models are too large to fit in on-chip memory on FPGAs, the weights have to be stored in off-chip memory. Chapter 4 investigates a novel blocking-batching strategy to optimize the throughput of large LSTM designs on FPGAs with a performance analysis based on LSTM models to balance trade-offs between area, power and performance. In addition, a stall-free hardware architecture is presented to eliminate the data dependencies and stalls, thereby further increasing the throughput of the design. Compared to the state-of-the-art design that stores the weights in off-chip memory, our approach achieves 1.65 times higher

performance-per-watt efficiency and 1.60 times higher performance-per-DSP efficiency. When compared with CPU and GPU implementations, our novel hardware architecture is 23.7 and 1.3 times faster while consuming 208 and 19.2 times less energy, respectively. This work has been published in papers [30, 31].

1.2.3 Low Latency RNNs with Partially-folded Architectures

The third contribution of this thesis is a low latency design of RNNs on FPGAs, addressing Challenge C3 and Objective O3.

Chapter 5 presents a partially-folded architecture for RNN, which maps all the layers on-chip and performs computation on their own units with dedicated optimization to achieve low latency and high throughput. This chapter also introduces a technique for balancing IIs in multi-layer RNN inferences, improving hardware efficiency and increasing design throughput. Additionally, a low latency LSTM template is devised, which enables the generation of low-latency FPGA designs with efficient resource utilization by HLS tools. We have open-sourced the template with some examples.[1] This work has been published in papers [32, 33]. The balancing II technique has also been discussed in papers [34, 35].

1.2.4 Connection between the Contributions

Fig. 1.6 illustrates how the three contributions of this thesis link together. Chapters 3 and 4 respectively present optimizations for accelerating RNNs with weights in on-chip memory and off-chip memory on a single FPGA, as shown in the bottom left of Fig. 1.6. The optimizations in these chapters are based on a fully-folded hardware architecture, as shown in Fig. 1.4(a). The designs in Chapters 3 and 4 utilize all the computing resources to form a large-scale single physical engine that leverages data-level parallelism.

In general, it is possible to build a fully-folded architecture design by combining the novel features of Chapters 3 and 4 to take advantage of both chapters. In fact, the idea of column-

[1] https://github.com/walkieq/RNN_HLS

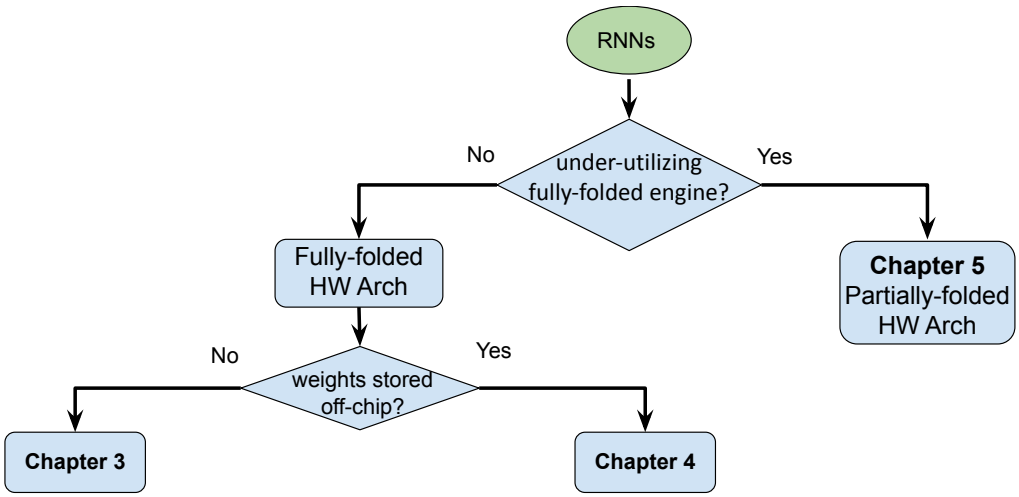


Figure 1.6: The connection between Chapter 3, 4 and 5. "HW Arch" stands for hardware architecture.

wise MVM (Chapter 3) has been adapted in the designs for Chapter 4. However, more work is needed to combine the blocking-batching strategy (Chapter 4) with the checkerboard tiling strategy (Chapter 3), as well as other hardware enhancements such as CAT units (Chapter 3). The details are included as future work in Chapter 6.

A hardware engine with a fully-folded architecture could be under-utilized when running a multi-layer RNN model because of its size, as discussed in Section 1.1.3. To address the issue, Chapter 5 introduces a partially-folded hardware architecture which can increase the hardware utilization and reduce the design latency, as shown in the right top of Fig. 1.6. The fully-folded engine is the engine with a fully-folded architecture. Investigating new hardware architectures and optimization techniques for RNNs on FPGAs, such as hybrid architectures that combine the fully-folded and partially-folded architectures, will be our future work.

Although this thesis focuses on RNNs, the fully-folded architecture and partially-folded architecture are general and could also be applied to other deep neural networks, such as convolutional neural networks (CNNs), graph neural networks (GNNs), transformer neural networks (TNNs), etc, resulting in various design options for various applications. We leave this as future work discussed in Chapter 6.

1.3 Selected Publications

The work in this thesis has been published in the following conference and journal papers.

The following publications contribute to Chapter 3:

- **Zhiqiang Que**, Hiroki Nakahara, Eriko Nurvitadhi, Hongxiang Fan, Chenglong Zeng, Jiuxi Meng, Xinyu Niu, and Wayne Luk. "Optimizing reconfigurable recurrent neural networks." In *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 10-18. IEEE, 2020.

- **Zhiqiang Que**, Hiroki Nakahara, Hongxiang Fan, Jiuxi Meng, Kuen Hung Tsoi, Xinyu Niu, Eriko Nurvitadhi, and Wayne Luk. "A reconfigurable multithreaded accelerator for recurrent neural networks." In *2020 International Conference on Field-Programmable Technology (ICFPT)*, pp. 20-28. IEEE, 2020.

- **Zhiqiang Que**, Hiroki Nakahara, Eriko Nurvitadhi, Andrew Boutros, Hongxiang Fan, Chenglong Zeng, Jiuxi Meng, Kuen Hung Tsoi, Xinyu Niu, and Wayne Luk. "Recurrent Neural Networks With Column-Wise Matrix-Vector Multiplication on FPGAs." *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 30, no. 2 (2021): 227-237.

The following publications contribute to Chapter 4:

- **Zhiqiang Que**, Thomas Nugent, Shuanglong Liu, Li Tian, Xinyu Niu, Yongxin Zhu, and Wayne Luk. "Efficient weight reuse for large LSTMs." In *2019 IEEE 30th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, vol. 2160, pp. 17-24. IEEE, 2019.

- **Zhiqiang Que**, Yongxin Zhu, Hongxiang Fan, Jiuxi Meng, Xinyu Niu, and Wayne Luk. "Mapping large LSTMs to FPGAs with weight reuse." *Journal of Signal Processing Systems* 92, no. 9 (2020): 965-979.

The following publications contribute to Chapter 5:

- **Zhiqiang Que**, Erwei Wang, Umar Marikar, Eric Moreno, Jennifer Ngadiuba, Hamza Javed, Bartłomiej Borzyszkowski, Thea Aarrestad, Vladimir Loncar, Sioni Summers, Maurizio Pierini, Peter Y Cheung, Wayne Luk. "Accelerating recurrent neural networks for gravitational wave experiments." In *2021 IEEE 32nd International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pp. 117-124. IEEE, 2021.

- Martin Ferianc, **Zhiqiang Que (Co-first author)**, Hongxiang Fan, Wayne Luk, and Miguel Rodrigues. "Optimizing Bayesian Recurrent Neural Networks on an FPGA-based Accelerator." In *2021 International Conference on Field-Programmable Technology (ICFPT)*, pp. 1-10. IEEE, 2021.

- **Zhiqiang Que**, Marcus Loo, Hongxiang Fan, Maurizio Pierini, Alexander D Tapper, Wayne Luk, "Optimizing Graph Neural Networks for Jet Tagging in Particle Physics on FPGAs." In *32th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2022.

The following publications are produced during my PhD research but they are not discussed in this thesis:

- **Zhiqiang Que**, Daniel Holanda Noronha, Ruizhe Zhao, Steven JE Wilton, and Wayne Luk. "Towards in-circuit tuning of deep learning designs." In *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 1-6. IEEE, 2019.

- **Zhiqiang Que**, Yanyang Liu, Ce Guo, Xinyu Niu, Yongxin Zhu, and Wayne Luk. "Real-time anomaly detection for flight testing using AutoEncoder and LSTM." In *2019 International Conference on Field-Programmable Technology (ICFPT)*, pp. 379-382. IEEE, 2019.

- Shijin Song, **Zhiqiang Que**, Junjie Hou, Sen Du, and Yuefeng Song. "An efficient convolutional neural network for small traffic sign detection." Journal of Systems Architecture 97, 269-277, 2019.

- Daniel Holanda Noronha, Ruizhe Zhao, **Zhiqiang Que**, Jeffrey Goeders, Wayne Luk, and Steve Wilton. "An overlay for rapid FPGA debug of machine learning applications." In *2019 International Conference on Field-Programmable Technology (ICFPT)*, pp. 135-143. IEEE, 2019.

- Hiroki Nakahara, **Zhiqiang Que**, Akira Jinguji, and Wayne Luk. "R2CNN: Recurrent Residual Convolutional Neural Network on FPGA." In *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 319-319. 2020. (Abstract).

- Hiroki Nakahara, **Zhiqiang Que**, and Wayne Luk. "High-throughput convolutional neural network on an FPGA by customized jpeg compression." In *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 1-9. IEEE, 2020.

- **Zhiqiang Que**, Daniel Holanda Noronha, Ruizhe Zhao, Xinyu Niu, Steven JE Wilton, and Wayne Luk. "Towards Overlay-based Rapid In-Circuit Tuning of Deep Learning Designs." In *2020 International Conference on Field-Programmable Technology (ICFPT)*, pp. 301-301. IEEE, 2020.

- Hongxiang Fan, Martin Ferianc, Shuanglong Liu, **Zhiqiang Que**, Xinyu Niu, and Wayne Luk. "Optimizing FPGA-based CNN accelerator using differentiable neural architecture search." In *2020 IEEE 38th International Conference on Computer Design (ICCD)*, pp. 465-468. IEEE, 2020.

- Daniel Holanda Noronha, **Zhiqiang Que**, Wayne Luk, and Steven JE Wilton. "Flexible Instrumentation for Live On-Chip Debug of Machine Learning Training on FPGAs." In *2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 20-28. IEEE, 2021.

- **Zhiqiang Que**, Daniel Holanda Noronha, Ruizhe Zhao, Xinyu Niu, Steven JE Wilton, and Wayne Luk. "In-circuit tuning of deep learning designs." *Journal of Systems Architecture* 118 (2021): 102198.

- Hongxiang Fan, Shuanglong Liu, **Zhiqiang Que**, Xinyu Niu, and Wayne Luk. "High-Performance Acceleration of 2-D and 3-D CNNs on FPGAs Using Static Block Floating Point." *IEEE Transactions on Neural Networks and Learning Systems* (2021).

- **Zhiqiang Que**, Marcus Loo, and Wayne Luk. "Reconfigurable Acceleration of Graph Neural Networks for Jet Identification in Particle Physics." In *2022 IEEE 4th International Conference on Artificial Intelligence Circuits and Systems (AICAS)*, pp. 202-205. IEEE, 2022.

- Hongxiang Fan, Martin Ferianc, **Zhiqiang Que**, He Li, Shuanglong Liu, Xinyu Niu, and Wayne Luk. "Algorithm and Hardware Co-design for Reconfigurable CNN Accelerator." In *2022 27th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pp. 250-255. IEEE, 2022.

- Hongxiang Fan, Martin Ferianc, **Zhiqiang Que**, Xinyu Niu, Miguel Rodrigues, and Wayne Luk. "Accelerating Bayesian Neural Networks via Algorithmic and Hardware Optimizations." *IEEE Transactions on Parallel and Distributed Systems* (2022).

- **Zhiqiang Que**, Hiroki Nakahara, Hongxiang Fan, He Li, Jiuxi Meng, Kuen Hung Tsoi, Xinyu Niu, Eriko Nurvitadhi, and Wayne Luk. "Remarn: A Reconfigurable Multi-threaded Multi-core Accelerator for Recurrent Neural Networks." *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* (2022).

- Hongxiang Fan, Martin Ferianc, **Zhiqiang Que**, Shuanglong Liu, Xinyu Niu, Miguel Rodrigues, and Wayne Luk. "FPGA-based Acceleration for Bayesian Convolutional Neural Networks." *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2022).

- Qianzhou Wang, Yat Wong, **Zhiqiang Que**, and Wayne Luk. "Verifying Hardware Optimizations for Efficient Acceleration." In International Symposium on Highly-Efficient Accelerators and Reconfigurable Technologies, pp. 17-23. 2022.

- Markus Rognlien, **Zhiqiang Que**, Jose G. F. Coutinho, and Wayne Luk. "Hardware-Aware Optimizations for Deep Learning Inference on Edge Devices." In *International*

Symposium on Applied Reconfigurable Computing (ARC), 2022.

- Filip Wojcicki, **Zhiqiang Que**, Alexander D Tapper, and Wayne Luk. "Accelerating Transformer Neural Networks on FPGAs for High Energy Physics Experiments" in *2022 International Conference on Field-Programmable Technology (ICFPT)*. IEEE, 2022.

- **Zhiqiang Que**, Shuo Liu, Markus Kongstein Rognlien, Ce Guo, Jose G De Figueiredo Coutinho, and Wayne Luk. "MetaML: Automating Customizable Cross-Stage Design-Flow for Deep Learning Acceleration." In *33th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2023.

- **Zhiqiang Que**, Hongxiang Fan, Marcus Loo, Michaela Blott, Maurizio Pierini, Alexander D Tapper, and Wayne Luk. "LL-GNN: Low Latency Graph Neural Networks on FPGAs for High Energy Physics." 2023. (Under review).

Chapter 2

Background and Related Work

2.1 Background

2.1.1 Recurrent Neural Network (RNN)

RNNs are ANNs with loops in them. The output of the previous nodes in RNNs forms part of the input to the subsequent nodes, allowing information to persist. They have feedback connections to retain past information about long-term dependencies over an arbitrary time. Before RNNs, the information about the correlation between the current data and previous data does not contribute to the training of ANNs since each data is considered independent of any others. However, there are many tasks where the current status highly depends on both current data and previous data. Therefore, the RNNs with feedback connections are used, which can take the previous information into account for the present decision.

A basic structure of a RNN is shown in Fig. 2.1. The x_t represents the input at the timestep t while the h_t represents the output, and A is a node or layer of neural network. Fig. 2.1(left) shows the rolled representation and emphasizes the recurrent nature of RNNs. The loop with A node allows information to be passed from one step to the next step. This can be seen as multiple copies of the same network, each passing a message to a successor. Fig. 2.1(right) shows the time-step unrolled representation. The unrolled RNN is a feedforward neural network

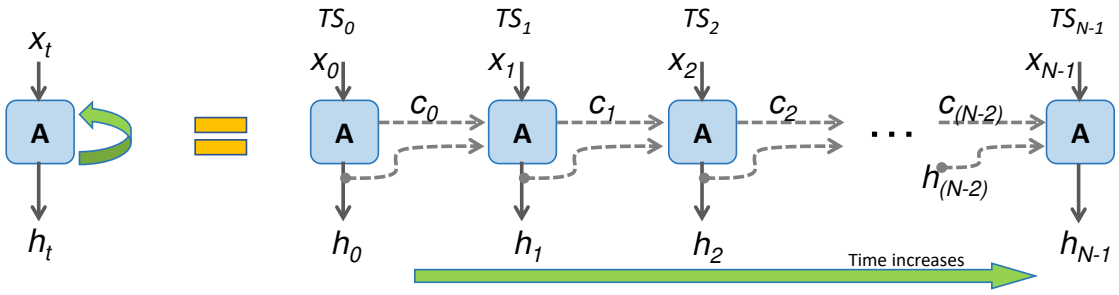


Figure 2.1: A recurrent neural network with time-step unfolded.

using the same weights and bias throughout the unrolled layer, appearing at each time step. The inputs and the outputs in the unfolded representation are the inputs x and outputs h at the timesteps 0, 1, 2 and N. The equation below are the calculations which take place in a simple RNN timestep t:

$$h_t = tanh(W_x x_t + W_h h_{t-1} + b) \tag{2.1}$$

where W_x and W_h represent the weights for the input vector and hidden vector respectively. b represents bias and $tanh$ represents hyperbolic tangent function which is an activation function.

To train an RNN, standard backpropagation is unsuitable due to the recurrence. Backpropagation Through Time (BPTT) [36], a variation of backpropagation, is often used to update the weights of an RNN, taking into account the times-step unrolling of RNNs. In BPTT, it needs to conduct many products when updating the weights of an RNN with lots of timesteps. This can create many small values due to the chain rule of partial derivatives to calculate the gradient. These products eventually get very close to zero, preventing RNNs from learning more information, which is known as the vanishing gradient problem [37]. This problem makes it difficult for vanilla RNNs to learn long-term dependencies. In some sequences, earlier inputs can influence inputs much later in the sequence, making the vanilla RNNs struggling to interpret such sequences.

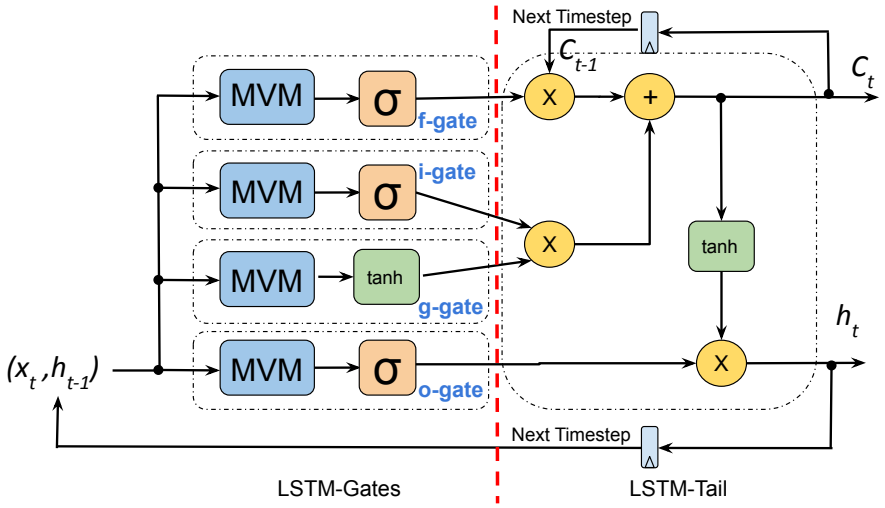


Figure 2.2: A detailed diagram of an LSTM Cell with loop-carried dependence.

2.1.2 Long-Short Term Memory (LSTM)

Vanilla RNNs are difficult to train for long sequences due to vanishing gradients [37]. To address this issue, Hochreiter *et al.* introduced Long-Short Term Memory (LSTM) networks [13]. The LSTM networks are a type of Recurrent Neural Networks (RNN) which relies on a memory controller to learn long-term dependencies. Since it is introduced, there have been many modifications to the original LSTM cell for different applications, but the changes to the standard architecture are minimal and their effects on the overall prediction accuracy are negligible.

This study follows the standard LSTM cell [1, 2, 11, 38]. The detailed structure of an LSTM cell or the node A mentioned in Fig. 2.1 is illustrated in Fig. 2.2. It utilizes the following equations to compute the gates and produce the results for the next time step.

$$\begin{aligned} i_t &= \sigma(W_i[x_t, h_{t-1}] + b_i), & f_t &= \sigma(W_f[x_t, h_{t-1}] + b_f) \\ g_t &= \tanh(W_g[x_t, h_{t-1}] + b_u), & o_t &= \sigma(W_o[x_t, h_{t-1}] + b_o) \\ c_t &= f_t \odot c_{t-1} + i_t \odot g_t, & h_t &= o_t \odot tanh(c_t) \end{aligned} \tag{2.2}$$

Here, σ and $tanh$ represent the sigmoid function and hyperbolic tangent function. Both are activation functions. i_t, f_t, g_t and o_t stand for the output of the input gate (i-gate), forget gate (f-gate), input modulation gate (g-gate) and output gate (o-gate) at timestep t respectively.

The g-gate is often considered as a sub-part of the i-gate. Each LSTM gate consists of a MVM unit and the addition of bias as well as a corresponding activation function unit, as shown in Fig. 2.2. The $\odot$ operator denotes an element-wise multiplication. W is the weight matrix for both input and hidden units since the input vector and hidden vector are combined in the equations. The b terms denote the bias vectors. c_t is the internal memory cell status at timestep t while h_t is the hidden vector which is the output of the cell and passed to the next timestep calculation or next LSTM layer.

The LSTM information flow is controlled by these four gates with details shown in Fig. 2.2. The i-gate decides what new information is to be written into the internal memory cell; the g-gate modulates the information processed by i-gate via adding non-linearity. Note that only g-gate utilizes hyperbolic tangent as its activation function while all the other three gates utilize sigmoid. The f-gate decides what old information is no longer needed and can be discarded so there are element-wise multiplications between the output of f-gate and memory cell status in the previous timestep c_{t-1}. Its output will be added to the products of the outputs from i-gate and g-gate to form the current status of the internal memory cell. The o-gate decides what the value of the current hidden vector (h_t) should be by multiplying the current status of the memory cell after the hyperbolic tangent function, as shown in the LSTM-Tail in Fig. 2.2. Our work focuses on the optimization of RNN inferences involving standard LSTMs, but the proposed techniques can be applied to other Deep Neural Network (DNN) inferences.

Gated Recurrent Unit (GRU), introduced by Cho *et al.*[39] in 2014, is a variant of LSTM. It combines the forget and input gates into a single "update gate" which determines how much of the previous hidden state to keep and how much of the new input to add. As a result, the GRU has one less gate than the LSTM, which leads to fewer parameters. Both LSTM and GRU have been shown to perform well in various applications, and in some cases, the GRU may outperform LSTM, while in other cases, the opposite may be true [40, 41, 42, 43]. However, the LSTM was introduced several decades before the GRU, which allowed it to become more widely recognized and adopted in research. In addition, LSTM's effectiveness has been well studied and demonstrated in a wide range of tasks, which has further contributed to its popularity.

2.1.3 Data Dependencies in RNNs

RNN computations involve both inter-layer and intra-layer data dependencies. The latter can be further divided into inter-timestep and intra-timestep dependencies. Inter-layer data dependencies mean that the input of each layer is derived from the previous layer's output. Intra-timestep dependencies imply that the operations in a single timestep are dependent on previous operations. In feed-forward only neural networks like CNNs, where there are no timesteps, the intra-timestep are equivalent to intra-layer data dependencies. While two of the three data dependencies can be found in feed-forward neural networks, only RNNs have the inter-timestep data dependencies.

In RNNs, the output at each time step depends on the previous output and the current input. This creates a dependency between the data at different time steps (inter-timestep), as the network relies on this information to make predictions. For example, in a language model, the RNN takes a sequence of words as input and predicts the next word in the sequence. The prediction at each time step depends on the words that came before it, as well as the current input word. This creates a dependency between the data at different time steps, as the network uses the information from previous time steps to make its prediction.

Algorithm 2.1 illustrates the pseudocode of an LSTM layer. Wx and Wh denote the LSTM weights for input and hidden vectors. B represents the bias. TS is the timestep. x is a set of input vectors and has the size of $(TS,\ Lx)$. h is a set of hidden vectors and has the size of $(TS,\ Lh)$. Seq decides if the whole sequence of hidden vectors should be returned or just the one in the last timestep.

The function $MVM\_x()$ performs MVM operations and the addition of bias for the LSTM gates involving the input vectors. The MVMs involving the hidden vectors are conducted in the function $MVM\_h()$. The $Sigmoid\_tanh()$ is the activation function which performs sigmoid or hyperbolic tangent operations. The $LSTM\_tail()$ function contains the element-wise operations as shown in Fig. 2.2. The result h_t as labeled in red is required in $MVM\_h()$ in the next timestep iteration, which shows the existence of data dependencies between different

Algorithm 2.1: The pseudocode of an LSTM layer.

```
1  Function LSTM_layer(Wx, Wh, B, x, Seq):
2      h_0 ← 0;
3      for t = 1 to TS do
4          Acc = MVM_x (Wx, B, x_t);
5          Acc = MVM_h (Wh, Acc, h_{t-1});
6          Acc = Sigmoid_tanh (Acc);
7          h_t = LSTM_tail (Acc);
8      if Seq then
9          return h;                          ▷ A set of all the hidden vectors
10     else
11         return h_t;                        ▷ The hidden vector at the final timestep
12 End Function
```

time-steps. As [1] mentions, RNN programs have a critical loop-carry dependence on the h_t vector. If the full pipeline of the hardware accelerator cannot return h_t to the vector register file in time to start the next timestep then the hardware kernel will stall.

This thesis proposes several techniques that can alleviate this problem. In chapter 3, we propose to calculate the MVMs column-wisely to alleviate the data dependencies, resulting in good performance. In Chapter 5, we propose to split LSTM unit into two sub units based on the data dependencies, and reallocate hardware resources to balance the initiation interval to achieve good hardware performance.

2.1.4 AutoRegressive Integrated Moving Average (ARIMA)

One of the successful applications using LSTMs/RNNs is forecasting time series. Traditionally, there are some statistical methods that can effectively forecast the next lag of time series data. The most well known method is AutoRegressive Integrated Moving Average (ARIMA) which uses a mathematical model to describe the relationship between the current value of a time series and its past values.

LSTM and ARIMA are two different approaches for modeling and forecasting time series data. There are many differences between them. The key differences are: first, LSTM is a type of neural network that uses a deep learning approach involving training and inference steps,

while ARIMA does not have a training step. Second, LSTM has more hyper-parameters than ARIMA and can model more complex time series. But it also means LSTM is more hard to tune compared with ARIMA. Third, LSTM does not require the data to be stationary but stationarity is important assumption in ARIMA.

In summary, LSTM is more powerful but also more complex and more difficult to tune, while ARIMA is simpler and easier to use but may not fit all types of time series tasks [44, 45]. Highlighting these differences helps to provide a broader perspective on time series forecasting methods. Furthermore, the discussion on ARIMA underlines the fact that different tools may be preferable under different conditions, promoting a comprehensive understanding of time series forecasting.

2.1.5 Field-Programmable Gate Array (FPGA)

A Field-Programmable Gate Array (FPGA) is a type of integrated circuits that can be re-configured to perform any digital function as a physical circuit after manufacturing. It has an array of configurable logic elements and programmable interconnects, allowing it to be tailored for specific digital circuit design requirements. The basic building block of FPGAs is Look-Up Table (LUT) which is implemented using small Random-Access Memories (RAMs). The LUTs implement combinational logic by storing a truth table and accessing it with the logic inputs serving as the address. To implement sequential circuits, FPGAs utilize registers in conjunction with LUT outputs. With a large number of LUTs and registers, FPGAs can support the implementation of large-scale parallel circuits. In addition, modern FPGAs also have coarse-grained resources that provide high level functionalities fixed in silicon, such as multipliers, digital signal processing (DSP) blocks, on-chip RAM, and even built-in processor cores. These processor cores can either be dedicated hard processors, such as the ARM processors found in Intel Arria [46] or Xilinx Zynq [47] FPGAs, or created from LUTs to form soft processor cores, such as Intel's Nios V [48] and AMD's MicroBlaze [49]. There are different sizes of FPGAs available, each offering different amounts of programmable logic resources. The larger ones provide more resources, which allows for more parallel circuits and higher levels of acceleration.

Designers have the option to choose from various FPGAs with different cost and performance trade-offs.

The widespread use of machine learning, particularly in the form of DNN networks, has led to a significant rise in computational requirements. To address this, AMD has developed AI Engines [50] which are optimized for linear algebra, provide the compute density to meet these demands. These engines are structured as 2D arrays consisting of multiple AI Engine tiles, optimized for real-time machine learning computations with deterministic performance. Intel has a similar solution, known as AI Tensor Blocks [51] found in Stratix 10 FPGAs [52] and the next generation Agilex devices [53], which have dense matrix math units optimized for 8-bit and 4-bit integer operations and mixed precision computations.

FPGAs have several features that make them well-suited for implementing accelerators for RNNs. FPGAs can offer large amounts of parallelism, which can be leveraged to perform multiple computations in parallel, thereby reducing latency for RNNs. In addition, they include high-performance DSP blocks that are able to perform mathematical operations effectively, such as MVMs, which are required for RNN computations. Moreover, FPGAs can be reprogrammed after manufacturing, making it possible to quickly modify the design of an RNN accelerator with optimized circuits specifically designed to adapt to changing requirements, resulting in good performance and efficiency.

2.2 Related Work

We discuss the related work addressing the challenge C1, C2 and C3 in Section 2.2.1, 2.2.2 and 2.2.3, respectively. We also discuss the acceleration of cloud-based RNNs in Section 2.2.5, targeting the same application domain as Chapter 3. Finally, exploiting the sparsity is an important topic in accelerating RNNs, which is discussed in Section 2.2.4.

2.2.1 Alleviating Data Dependencies for RNNs on FPGAs

There have been many previous studies on alleviating data dependencies for RNNs running on FPGAs [1, 2, 15, 16, 19, 21, 22, 54, 55, 56, 57, 58]. The Brainwave design[1] is a single-threaded SIMD Instruction Set Architecture (ISA) primarily comprised of matrix-vector and vector-vector operations for running real-time artificial intelligence (AI), including persistent RNNs. It leverages Vector-Level Parallelism (VLP), where compound operations are partitioned into smaller operations with a fixed native vector size. Besides, it connects vector functional units in a dataflow architecture, enabling vectors to flow directly from one functional unit to another, thereby minimizing pipeline bubbles. One of the key features of the Brainwave architecture is the use of a technique called "instruction chaining" which helps to mitigate data dependencies and improve efficiency. It allows sequences of dependent instructions to pass values directly from one operation to the subsequent one. As opposed to conventional methods, this explicit chaining mitigates data dependencies and enables the microarchitecture to exploit substantial pipeline parallelism, bypassing the need for complex hardware dependency checking or multi-ported register files, leading to an efficient and streamlined architecture for RNNs. The Brainwave-like NPU [2] is based on the same technique to handle data dependencies. C-LSTM [54] breaks down the LSTM pipeline into several smaller coarse-grained pipelines and overlaps their execution time with a directed acyclic data dependency graph representing the computation flow of LSTM. But these designs only consider the intra-timestep data dependencies, as shown in Table 2.1.

RNNs have a lower degree of parallelism compared to CNNs because of the inter-timestep dependencies [1]. RNNs process sequences of inputs, where the output at each time step depends on the previous time steps, resulting in a dependency between time steps which prevents parallel computation of different input vectors. The batch technique [15, 16, 19, 21, 22, 56, 57] is the most common used technique to alleviate data dependencies in RNNs by interleaving different input inference requests, as there are no data dependencies in different input inference requests. However, using a batch of inputs can harm latency, as discussed in Section 1.1. The recently introduced SHARP [58] (2022) is a specialized and adaptable ASIC-based architecture specif-

Table 2.1: Summary of previous designs that alleviate data dependencies for RNNs on FPGAs

	Addressing Data Dependencies			Weights Storage
	intra-timestep	inter-timestep	techniques	
Brainwave [1]	✓	×	Instruction Chaining	on-chip
NPU [2]	✓	×	Instruction Chaining	on-chip
C-LSTM [54]	✓	×	Coarse-grained Pipeline	off-chip
[55]	✓	×	3-stage Scheduling	off-chip
[15, 16, 22, 56]	✓	✓	Batching	on-chip
FP-DNN [19]	✓	✓	Batching	off-chip
[57, 21]	✓	✓	Batching	off-chip
SHARP [58]	✓	✓	Unfolded Scheduling	on-chip
Chapter 3	✓	✓	Column-wise MVM & Pipeline	on-chip
Chapter 4	✓	✓	Blocking-Batching Strategy	off-chip

ically tailored for RNNs. It leverages a combination of unfolded scheduling that unfolds the MVM of the input and hidden vectors in order to alleviate inter-timestep data dependencies, and a dynamically reconfigurable compute engine that optimizes resource mapping. To handle the data dependencies, it pre-calculates the next step's input MVM and stores the partial results in an intermediate buffer when the engine is processing the last sequential computation of the cell state and hidden outputs for the current timestep. Besides, the reconfigurable compute engine allows the accelerator to adapt to different RNN configurations and handles the padding issue caused by matrix-vector multiplication more effectively. However, the original paper only provides a high-level overview of the design, and it does not have details of the microarchitecture implementations.

Chapter 3 describes a novel hardware architecture for RNNs, which is based on the column-wise MVM to alleviate data dependencies, as well as several other optimization techniques to improve the design's performance and efficiency. The Chapter 4 combines the column-wise MVM and blocking-batching strategy to reuse the weights to alleviate the inter-timestep data dependencies for RNNs with weights on the off-chip memory.

2.2.2 Acceleration of RNNs with Weights Off-chip

There are also many previous studies about LSTM implementations with weights stored in off-chip memory on FPGAs [19, 20, 22, 30, 38, 59, 60, 61, 62]. This has been recognized as a performance bottleneck due to the large latency incurred by repeatedly loading the weights matrix from off-chip memory when the size of on-chip memory is not large enough to store the entire matrix. Chang *et al.* [59] present an FPGA-based hardware implementation of LSTM on the Xilinx Zynq 7020 with 16-bit quantization for weights and input data, both of which were stored in off-chip memory. Guan *et al.* [38] propose a smart memory organization with on-chip double buffers to overlap computations with data transfers. Later in [19], they present FP-DNN, an end-to-end automated framework that maps Convolutional Neural Networks (CNNs) or RNNs on FPGAs with RTL-HLS hybrid templates. FP-DNN maps LSTM inferences to matrix-matrix multiplication kernels. However this is inefficient since there are only matrix-vector multiplications in LSTMs. Therefore, they batch the input requests to convert the matrix-vector multiplication to matrix-matrix multiplication to improve the performance of LSTMs in the FP-DNN framework. Besides, their work does not explore the data dependencies issue of LSTMs. Other studies such as [18, 20, 21] apply the batching technique to increase the throughput of LSTM inferences. For example, E-BATCH is proposed [21] for RNNs, which improves throughput and energy efficiency on an ASIC-based accelerator. By combining the input vectors from various requests for the same model, the weights can be reused among them, resulting in a good communication-to-computation ratio. However, this batch technique can harm latency because different requests may not arrive at the same time [20], which means that a newly arrived request must wait until the batch is formed, resulting in a latency penalty. Our chapter 4 covers the published work [30, 31], which introduce the blocking-batching strategy to increase the parallelism of RNNs while still targeting a single inference request, by exploiting the parallelism between various timesteps in a single RNN inference. This work inspires some subsequent studies [60, 61] by others in the community. [60] introduces a weight reuse scheme, called Time-Step Interleaved Weight Reuse (TSI-WR), which interleaves the computations of two adjacent time-steps in an LSTM cell. This interleaving allows for partial reuse of the on-chip weight matrices across the two time-steps, reducing the need for repeated accesses to off-chip

memory, leading to low power consumption. [61] introduces Split And Combine Computations (SACC) approach which enables weight reuse by dividing weight matrix into lower-diagonal, diagonal elements and upper-diagonal elements. These elements are accessed and reused across consecutive time steps during the computations, reducing the need for repeatedly accessing the same weights from off-chip memory. These two studies present fine-grained weight reuse techniques and reduce off-chip memory access, aiming to reduce power consumption.

2.2.3 RNN Accelerators with Partially-folded Architectures

Previous FPGA-based RNN accelerators have mostly focused on fully-folded hardware architectures. Only a few efforts have explored the use of partially-folded hardware architectures for RNNs. For example, Peng *et al.* [25] present a dual-engine accelerator for LSTMs, which can execute multiple RNN inferences simultaneously or have cores collaborate on a single inference. Khoda *et al.* [63] introduce ultra-low latency RNNs on FPGAs for physics applications based on partially-folded architectures using the HLS4ML tool, demonstrating the potential of low latency RNNs in scientific applications. This thesis provides a comprehensive analysis of fully-folded architectures and partially-folded architectures for RNNs on FPGAs. In addition, an II balancing technique is also introduced to balance the IIs among multiple engines, improving hardware efficiency.

Some other efforts have used multiple FPGAs as multiple engines to accelerate RNNs. [64, 65] introduce a multi-FPGA approach for accelerating multi-layer RNNs, with each FPGA processing an RNN layer on an FPGA-based cluster. The computation is unfolded along the layer axis but not the timestep axis, similar to the designs presented in Chapter 5. But Chapter 5 also introduces a technique to balance IIs, improving hardware efficiency and increasing design throughput. [66] explores various partitioning strategies of large RNN inferences, including single-layer RNN networks, to achieve scalable multi-FPGA acceleration. In contrast, this thesis focuses on a single chip implementation. However, the proposed partially-folded architecture can be easily extended to support multiple FPGAs with different engines on different FPGAs, each processing one layer or a few layers.

2.2.4 Acceleration of Sparse RNNs

There is also much previous work [9, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76] on exploiting the sparsity in data and weights through pruning to reduce computation in networks and also to reduce the memory footprint in order to achieve high performance and efficiency. ESE [9] proposes a load-balance-aware pruning method with an automatic dynamic-precision data quantization flow for LSTMs and compresses a large LSTM model by 20× without sacrificing prediction accuracy. The load balancing refers to the even distribution of non-zero weights (the computational tasks) across multiple Processing Elements (PEs). When a network is pruned, it becomes sparse, with non-zero weights representing the workload. If one PE has more non-zero weights than others, it creates a bottleneck, slowing down overall processing. The proposed load-balance-aware pruning aims to ensure a balanced distribution of non-zero weights, improving hardware performance and efficiency. It also includes a scheduler that partitions the compressed model for parallel processing and schedules LSTM operations to overlap memory references with computation. DeltaRNN [68] is a pruning method that exploits temporal sparsity in RNNs by skipping unnecessary computations and memory access using the delta network algorithm. It only updates the output of a neuron when the neuron's activation changes by more than a set threshold (delta). This leads to a speedup of up to 5.7 times with little loss in accuracy. However, it only supports a single layer GRU network. BBS (Bank-Balanced Sparsity) [70] is a structure pruning technique that divides weight matrix rows into banks for parallel computing and adopts fine-grained pruning within each bank to maintain model accuracy. It achieves both high prediction accuracy of a pruned LSTM model and high hardware efficiency of the model running on FPGAs. BLINK [73] designs the LSTM inference using a bit-sparse data representation. It converts multiplications into bit-shift operations to improve the energy efficiency while maintaining LSTM inference accuracy for real-time calcium image processing. The extension [77] proposes a combination of bit-sparse quantization and pruning methods for energy-efficient LSTM inferences. More recently, Spartus [75] exploits spatio-temporal sparsity to achieve ultra-low latency inference using Column-Balanced Targeted Dropout (CBTD). These studies are orthogonal to our proposed approach and hardware architecture. These techniques can be complementary to our approaches to achieve even higher performance and

efficiency of RNN/LSTM inferences using FPGAs.

2.2.5 Acceleration of Cloud-based RNNs

The Brainwave design [1] is a single-threaded SIMD architecture for running real-time artificial intelligence (AI), including persistent RNNs. It achieves more than an order of magnitude improvement in latency and throughput over state-of-the-art GPUs on large memory intensive RNN models at a batch size of 1. It stores NN model weights on-chip for RNNs to get a necessary high memory read bandwidth to achieve higher performances. A coarse-grained reconfigurable architecture (CGRA) based RNN accelerator is proposed [3] on top of Plasticine [78] with a set of techniques for performing cross-kernel optimization in RNN cells. AERO [79] is a single-threaded instruction-set based processor using a versatile vector-processing unit customized for RNN inferences on resource-limited FPGAs. A Brainwave-like neural processing unit (NPU) is proposed in [2] with a single-threaded architecture. They also explore the potential of combining a TensorRAM with FPGAs to provide large high-speed memory for large memory intensive RNN sequence models. Besides, their late work [22] deploys the Brainwave-like NPU on the Stratix 10 NX which is Intel's new AI-optimized FPGA featured with AI tensor blocks. Our chapter 3 and the related published work [24, 29] propose a novel latency-hiding hardware architecture based on column-wise MVM and fine-grained tiling strategy for cloud-based RNNs with good performance and power efficiency. Subsequent work [80, 81] presents RNN accelerators with spatial and temporal co-execution of multiple RNN/LSTM inferences, leading to good performance.

2.2.6 GPUs and other Commercial Chips for LSTMs

A Graphics Processing Unit (GPU) is a specialized processor designed to handle the graphical and mathematical computations required for rendering images and videos in real-time. They have become increasingly popular for general-purpose computations as well, especially in scientific computing and machine learning, due to their ability to perform many calculations in

parallel. There are many high-level frameworks, such as TensorFlow [82] and PyTorch [83], that offer built-in support for GPU acceleration, making it easy to use GPUs for LSTMs. However, they often consume a large amount of power, which hinders their deployment on devices with limited power supplies. GPUs can offer high performance with large amount of parallelism when running batches of input requests but may not perform well when there is no batch due to the data dependencies in LSTMs. In comparison, FPGAs offer benefits of low latency and low power consumption [1, 2, 24, 29].

There are also some emerging commercial machine learning chips that are used to accelerate LSTMs, such as Tensor Processing Unit (TPU) [84] from Google, Tensor Streaming Processor (TSP) [85, 86, 87] from Groq and SambaNova [88, 89]. The TPUs are developed by Google for accelerating machine learning workloads, which perform matrix computations efficiently. However, it suffers from low hardware utilization for LSTMs with an average utilization of 3.5% [84]. The Groq's TSP is a single processor and includes 409,600 multiply-accumulate units along with 5120 vector ALU units. They are organized into sets of superlanes, each executing a very long instruction word (VLIW) instruction. With a 1GHz frequency, its peak INT8 performance is around 820 TOPS. However, the maximum effective throughput is 20 TOPS for a batch=1 LSTM-512 (hidden vector size is 512) [87], resulting in 2.4% hardware utilization. SambaNova is based on Plasticine [78] architecture which is a coarse-grained reconfigurable architecture (CGRA). It achieves 4-5 times speedup for online inference performance of LSTM and GRU when compared with Nvidia's A100 [89]. A SambaNova-like CGRA specific for accelerating RNNs is proposed in [3], which is also on top of Plasticine [78]. The performance numbers comparing our designs and this SambaNova-like RNN accelerator are listed in Table 3.3. Our design (Large) achieves 1.8 times better performance than it [3].

2.3 Summary

This chapter presents background information that forms the foundation of this thesis. In addition, a review of various RNN designs on FPGAs is also presented. Many of these designs

address data dependencies in RNNs by stalls, which can lead to low design efficiency. Chapter 3 introduces several techniques to alleviate the data dependencies, resulting in good performance. Chapter 4 focuses on optimizing large RNNs with weights in off-chip memory. Chapter 5 presents a partially-folded hardware architecture for RNN, which maps all the layers on-chip with dedicated optimization for each layer to achieve low latency and high throughput. In summary, the following chapters of this thesis discuss various approaches to optimize reconfigurable designs for RNN acceleration, and achieve improvement in terms of design throughput, latency and power efficiency.

Chapter 3

Column-wise Matrix-Vector Multiplication for RNNs on FPGAs

3.1 Introduction

This chapter presents a reconfigurable accelerator for REcurrent Neural networks with fine-grained cOlumn-Wise matrix-vector multiplicatioN (RENOWN), involving a novel latency-hiding architecture which can eliminate data dependencies to improve the throughput of RNN inference systems.

To speed up RNN inferences, FPGAs have been utilized in various scenarios [1, 2, 38, 90], achieving lower latency and power consumption compared to CPUs and GPUs. However, the recurrent nature and data dependency in the RNN computation result in undesired system stall until the required hidden vectors return from the full pipeline to start the next time-step calculation [1]. Besides, while deep pipelining can be utilized to enhance operating frequency, it increases stall penalty due to longer drain time. Moreover, inefficient tiling can leave hardware resources idle resulting in low utilization. For example, the Brainwave [1] has six matrix-vector multiplication (MVM) "tile engines", each processing 400x40 matrices, so they have a peak capability of processing a 400x240 matrix in parallel. Any MVM that does not map to this dimension will leave some resources idle. Fig. 3.1 shows that Brainwave's hardware utilization

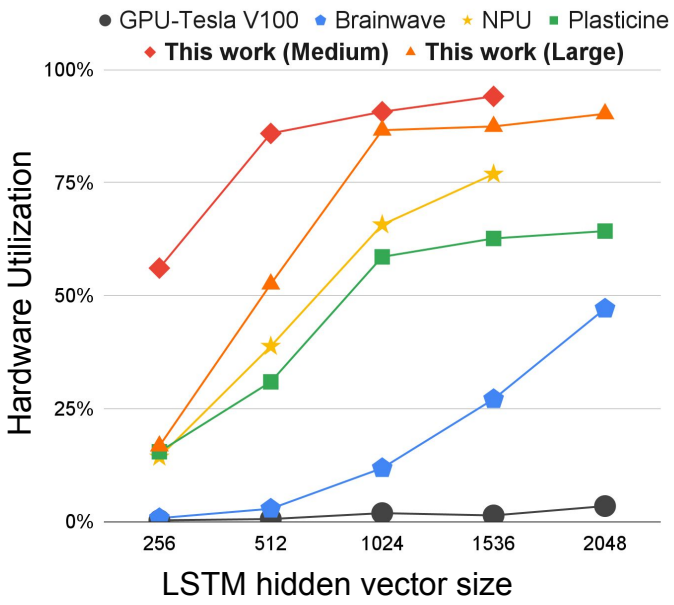
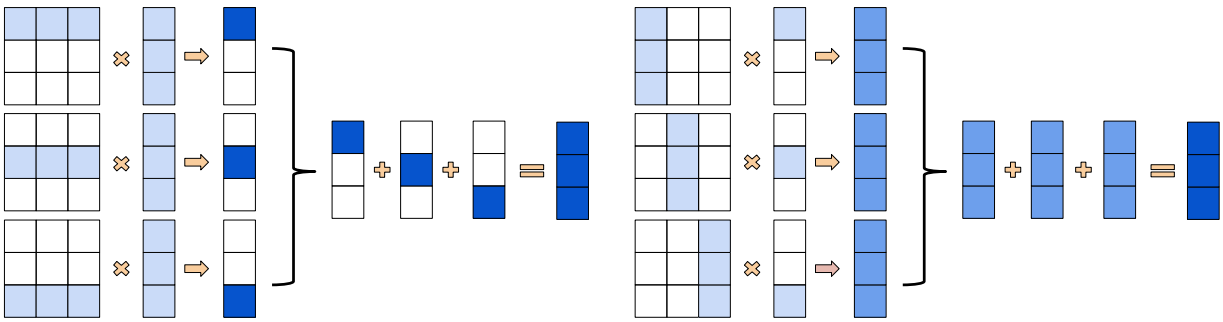


Figure 3.1: Hardware utilization of various LSTM implementations, including Nvidia Tesla V100 GPU, Brainwave [1], Intel Brainwave-like NPU [2], CGRA-based Plasticine [3] and our work. The hardware utilization is the proportion of hardware doing useful computation. These workloads are representative LSTM layers from popular DNN models such as DeepSpeech.

ranges from less than 1% to only about 50% for various LSTM models. Another implementation, Google's TPU, also suffers from low hardware utilization and achieves an average utilization of 3.5% for LSTMs [84]. The Brainwave-like Neural Processor Unit (NPU) [2] with a fine-grained zero-padding scheme achieves around 75% for a large LSTM. However, it still suffers from low utilization, especially from medium to small sized LSTMs which are commonly used in many applications [91, 38, 11]. LSTM models with small to medium sizes that have a large number of time-steps are the most tangible examples that require dealing with lots of dependencies, as well as the parallel task of MVMs [58]. With the need for high-performance systems, it is essential to maximize the hardware utilization to achieve the highest possible effective performance and energy-efficiency.

This work proposes a novel latency-hiding hardware architecture and a configurable checker-board tiling strategy for RNN/LSTM models to increase hardware utilization and enhance the throughput of RNN inference. First, we propose column-wise MVM for RNN/LSTM gates, which is able to eliminate their data dependencies. The column-wise block-striped decomposition of a matrix in MVM, as shown in Fig. 3.2b, is an effective outer-product based parallel



(a) Row-wise MVM. (b) Column-wise MVM (This work).

Figure 3.2: Matrix-vector multiplications (MVM).

method for processing MVM in high-performance computing. Recently there are also some outer-product based matrix-matrix multiplication accelerators [92, 93]. However, most of the previous FPGA-based RNN implementations focus on row-wise MVM as shown in Fig. 3.2a. The proposed architecture can start the calculation of the next time-step without waiting for the system pipeline to be drained, which means that the system can be fully pipelined without stalling.

Moreover, a novel configurable checkerboard tiling strategy is proposed which incorporates Element-based Parallelism (EP) and Vector-based Parallelism (VP) to boost inference throughput, as shown in Fig. 3.3. To support EP and VP, a new hardware architecture that supports hybrid kernels is proposed which combines multiplier-adder-tree and multiply-accumulate architectures. The architecture deploying many parallel multipliers followed by a large balanced adder-tree is commonly used in FPGA-based RNN/LSTM accelerators [1, 2, 15, 38]. These designs are based on row-wise MVM. To support MVM column-wise processing, our design deploys many parallel multipliers followed by accumulators, as shown in Fig. 3.2b. Furthermore, unlike our previous work [24] which is based on a fixed-size tiling approach, this work proposes a configurable tiling technique that supports various configurations of EP and VP to further improve the performance since different sizes of RNN models prefer different configurations of (EP, VP), as shown in Fig. 3.3

Our results indicate that the proposed acceleration architecture is not only the fastest compared to the state-of-the-art for a large LSTM model, but also much more suitable for a wider-range of RNN models in terms of complexity. Hence, it performs much better for RNN models with

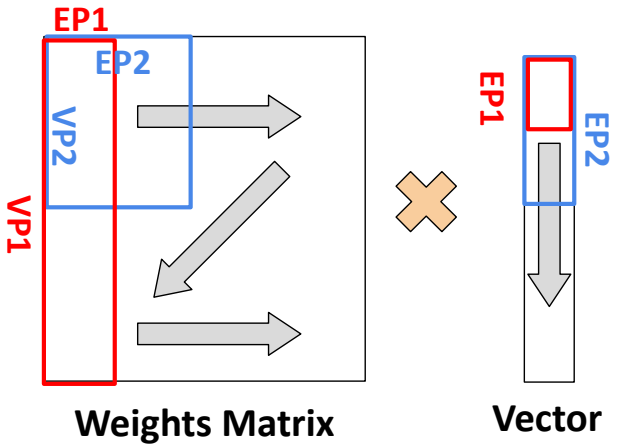


Figure 3.3: The Element-based Parallelism (EP) and Vector-based Parallelism (VP) with tiles shaded in red or blue. Two sets of (EP, VP) are shown in this example.

different sizes as shown in Fig. 3.1. We make the following contributions:

- Novel column-wise MVMs for RNNs to eliminate data dependencies, increasing the hardware utilization and system throughput.
- A flexible checkerboard tiling strategy supporting EP and VP to exploit the available parallelism while increasing hardware utilization. Besides, the (EP, VP) parameter space is comprehensively explored.
- A latency-hiding hardware architecture with novel hybrid kernels and Configurable Adder-tree Tail (CAT) units to support the proposed optimizations.

3.2 Design and Optimization

This section covers data dependency analysis and optimizations targeting RNN designs. Some system parameters are defined in Table 3.1.

Table 3.1: Summary of parameters used in this study

W	Weights matrix
w_n	All the weights of row n in W
w'_n	All the weights of column n in W
Hw	Number of Rows of weights matrix
Lw	Number of columns of weights matrix
x_t	The input vector x at timestep t
h_t	The hidden vector h at timestep t
$x_t[j]$	The element j in the input vector x at timestep t
$h_t[j]$	The element j in the hidden vector h at timestep t
Lx	Number of elements in the input vector x
Lh	Number of elements in the hidden vector h
NPE	Number of processing elements
EP	Element-based Parallelism
VP	Vector-based Parallelism
TS	Timestep
MVM_x	The MVM involving input vector x
MVM_h	The MVM involving hidden vector h

3.2.1 Weights Matrix of LSTM Gates

In this design, the four matrices of i, f, o, u gates in LSTMs are combined into one large matrix since they are of the same size. Thus, in one time-step calculation of the LSTM, we only need to focus on one large matrix multiplied by one vector for the whole LSTM cell instead of four small matrices multiplying one vector. This is a generic optimization that can be applied to any MVMs that share the same input vector. Since each gate matrix has the size of $Lh \times (Lx + Lh)$, the size of the combined matrix is $(4 \times Lh) \times (Lx + Lh)$. Then we have the $Hw = 4 \times Lh$ and $Lw = Lh + Lx$. Besides, the weights of the four LSTM gates are also interleaved in the final weights matrix. Therefore, the related elements in the result vector from four gates are adjacent and can be reduced easily via the element-wise operations in the LSTM-tail units.

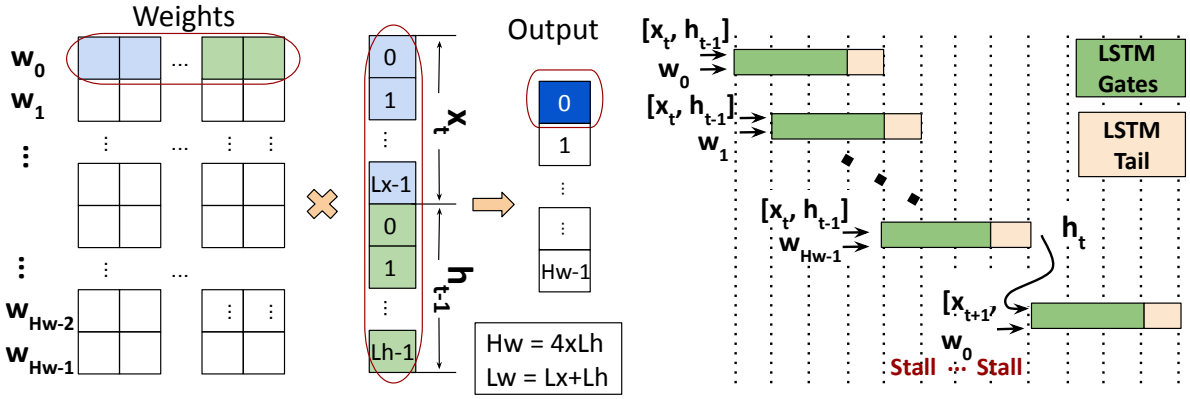
3.2.2 Row-wise MVM for RNNs

Conventional designs of MVM for RNNs are row-wise, and they have a major problem of being stalled when their pipelines are not fast enough to bring data back to the input for the next time step. They involve the entire vector of (x_t, h_{t-1}) and one or several entire rows of the weights matrix at a time, as shown in Fig. 3.4a. This approach imposes additional stalling since the system has to wait for a newly computed hidden vector before starting the calculation of next timestep.

Data hazard exists since the whole new hidden vector h_t is required to start the new computation of x_{t+1} in the conventional MVM design for RNN/LSTM. It is mainly due to the data dependencies between the output from the current timestep and the vector for the next timestep. It indicates that the whole system pipeline needs to be drained to get the new computed hidden vector h_t before the new matrix-vector operations can start. As [1] mentions, RNN programs have a critical loop-carry dependence on the h_t vector. If the full pipeline cannot return h_t to the vector register file in time to start the next timestep then the MVM unit will stall, as shown in Fig. 3.4b. On the other hand, deep pipelining is often required to achieve a high operating frequency for designs. This makes it difficult to achieve a design with the best trade-off.

3.2.3 The Proposed column-wise MVM for RNNs

This work proposes a new technique for calculating matrix-vector operations in a column-wise fashion. At the beginning, only a few elements from the x_t vector are used while h_{t-1} is not touched, but all the elements in the corresponding columns of the weights matrix are involved to perform the operations, as illustrated in Fig. 3.5a. To illustrate the idea, the number of the pipeline stages of the example system is 4 as shown in Fig. 3.5b. However, the pipeline stages of a real system can be much larger. In addition, the figure shows only one element in the x_t vector is used to perform the calculation for simplicity, but the actual number of the involved elements in each cycle depends on the architecture's parallelism requirements. The number of

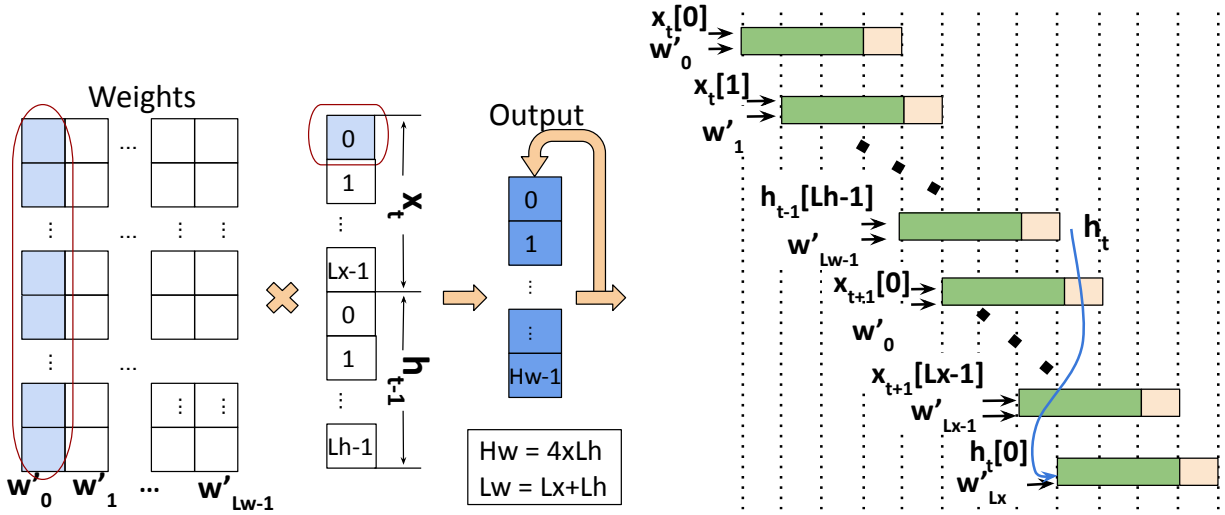


(a) Row-wise MVM for LSTMs (b) Pipeline analysis of (a)

Figure 3.4: The row-wise MVM with LSTM data dependencies analysis

elements employed in this work is EP which is explored and fine-tuned in Section 3.3. The partial result vector is generated from the small dot-product of the partial x_t vector and the corresponding weights. Then it is accumulated over multiple cycles to generate the final result vector. This way, the calculation of the new inference of (x_{t+1}, h_t) can start without waiting for the system pipeline to be drained to get h_t since it only needs a partial input vector. It indicates that the system can be fully pipelined without stalls, as shown in Fig. 3.5b. Each hidden vector can finish the calculation in the shadow region of processing x_t before it is required. The stall happens in the calculation of each timestep, and the total potential stalling cycles equals the design pipeline stages. When the LSTM workload is large, e.g., with a layer Lh as 2048, the number of processing cycles of such a workload will be much larger than the number of stall cycles and then the benefit of the column-based MVM will be small, because the ratio of the stall cycles to the processing cycles is small. However, when the workload is small, e.g., with a layer Lh as 256, the number of processing cycles of such a workload is also small. In such a scenario, reducing stall cycles is vital because the ratio of stalls is large. For example, if the number of processing cycles is the same as to the design pipeline stages, the hardware utilization can be increased from 50% to 100% if all the stalls can be hidden.

One disadvantage has been observed that although the column-wise MVM only needs a partial input vector, it produces the output vector later than the row-wise MVM, because it needs to wait for all the columns to be processed to get the final accumulated vector before producing the output [94]. It seems that the succeeding hardware units that depend on the output vector



(a) Column-wise MVM for LSTMs (b) Pipeline analysis of (a)

Figure 3.5: The column-wise MVM with LSTM data dependencies analysis

(e.g., those that perform activation functions and element-wise operations in the RNNs) would need to wait longer. In contrast, the row-wise MVM completes one subset of the output vector before moving to the next subset. Therefore, a subset of the final output vector is completed sooner than in a column-wise case. However, in the column-wise case, the succeeding units can get an entire output vector but not a subset. Although the column-wise architecture starts the subsequent processing later than the one using row-wise MVM, the number of succeeding units can be increased to match the output bandwidth and finish the whole calculation sooner than the row-wise case. Practically, we do not need to significantly increase the number of these units since they can process the whole vector over multiple cycles, until the next vector is produced by the MVM engine. Besides, the row-based approach may also involve multiple succeeding units to increase parallelism. Moreover, these units are much smaller compared to the MVM kernel engine that has lots of multipliers and adders. Thus, the extra hardware area is negligible compared to the whole accelerator.

When the size of x_t vector is small and/or the size of h_t vector is large, the design may still stall because the cycles of processing x_t vector cannot completely hide the whole pipeline latency to get the h_t ready before it is needed. However, with the column-wise MVM, we can still continue to process the MVM of x_t and its corresponding weights when we are waiting for the h_t to be computed. Besides, when the input vector is small, an LSTM model would rarely require a

significantly larger hidden vector.

3.2.4 Tiling and Parallelism

To further exploit the available parallelism, we introduce Element-based Parallelism (EP) and Vector-based Parallelism (VP) in our design, as shown in Fig. 3.3. The matrix of weights is split into small tiles with a size of (EP, VP). In each cycle, the hardware engine is able to process a tile of the weights matrix and a sub-vector of $[x_t, h_{t-1}]$ with a size of EP. EP and VP need to be determined carefully so that the number of cycles to process the x_t vector, given by $\frac{Lx}{EP}$, is larger than the system latency to ensure that the computation of hidden vectors can be fully hidden by processing x_t vector. This number is small when EP is large and it may still result in system stalls. To increase system parallelism, VP is chosen to be as large as possible. However, the largest number of VP is Hw, which equals $4 \times Lh$, since there are only 4 gates in LSTM. In summary, the hardware utilization and system throughput can be improved via balancing EP and VP.

A fixed configuration of (EP, VP) can bring low hardware utilization. Brainwave [1] has 6 MVM tile engines, each processing 400×40 matrices. The NPU [2] also has a fixed configuration of 4 tiles, 120-wide dot product engines and 40 lanes. Any MVMs that do not map well to these dimensions will leave some resources idle. Since RNNs are used for various tasks, RNN accelerators should support diverse configurations. This work proposes a novel hardware architecture to support various sets of (EP, VP) since models with different sizes may prefer different optimal EP and VP configurations. Fig. 3.3 shows the basic idea with two sets of (EP, VP).

One option is to adopt the row-wise MVM while cascading the computation of MVM_x and MVM_h, which are the MVMs involving input vector and hidden vector respectively, instead of a unified MVM. Cascading them with a row-wise fashion may also help to eliminate the data dependencies between current and next timestep calculations. However, this approach complicates the enhancement of parallelism. Usually the length of x and h are different, resulting in different computation loads for MVM_x and MVM_h. The input vector x is usually application

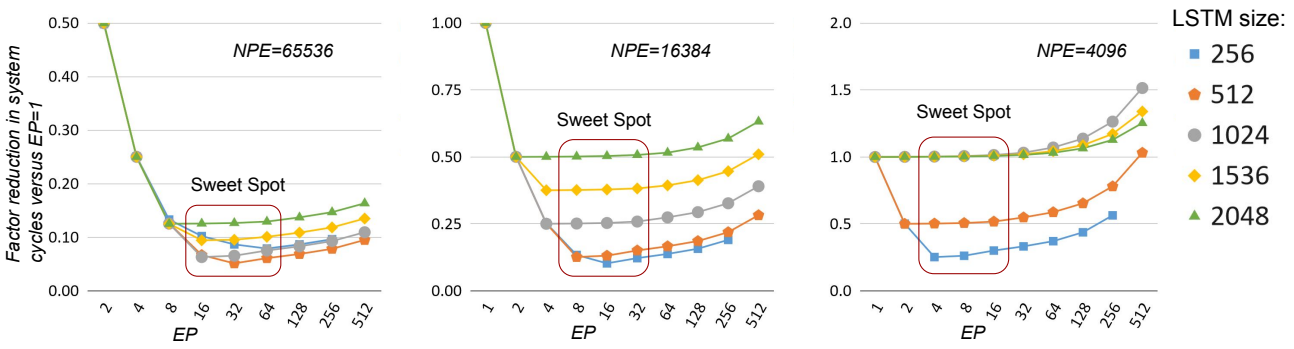


Figure 3.6: The number of processing cycles in our proposed column-wise approach for different values of EP, NPE, and model sizes. Different colors combining different point shapes represent different model sizes with Lh from 256 to 2048.

dependent while the value of h can be selected by designers to meet application requirements and to reduce hardware utilization. Zero padding may be required to support both MVM_x and MVM_h, which causes inefficiency. Actually the heights of the MVM_x and MVM_h are both $4Lh$, while the widths are Lx and Lh respectively. The column-wise based computation enables more parallelism than the row-wise one, which improves design efficiency. Both the designs [1, 2] separate and cascade the MVM_x and MVM_h, but they still suffer from low hardware utilization as explained above.

3.3 Design Space Exploration

The hardware design space is characterized by the tiling block size of (EP, VP) and the number of processing elements (NPE) after combining the configurations discussed previously. The effective performance varies with the tile size and the number of PEs. This design space exploration is independent of the particular FPGA technology.

To figure out the optimal parameters of the system configuration for our in-depth analysis, we develop a cycle-accurate simulator to conduct the design space exploration. A greedy algorithm is proposed to explore design space. It starts with $EP = 1$ while the VP is given according to

the system constraints shown in 3.1.

$$VP \leq 4 \times Lh \quad \text{and} \quad VP \leq \frac{NPE}{EP} \tag{3.1}$$

Practically EP and VP should be as large as possible because this would maximize the potential parallelism and the system throughput. However, when EP increases, the number of cycles needed for processing the input vector (Lx) decreases so that the system may not have sufficient cycles to completely hide the processing of the hidden vector as discussed in Section 3.2.3. In this exploration, the VP is set as large as possible, which is $min(4 \times Lh, \frac{NPE}{EP})$. Fig. 3.6 illustrates the exploration results for different sizes of LSTM models with Lh from 256 to 2048 with different colors and point shapes, using our hardware design when the NPE is 4096, 16384 or 65536. Fig. 3.7 shows the feasible sets of (EP, VP) when NPE is 65536. The number of processing cycles determines the throughput of the system and the lower it is, the better the overall performance will be. As shown in Fig. 3.6, when EP is small, the number of processing cycles is high because the VP is constrained by Equation (3.1) so that the number of effective PEs is less than NPE, which leads to severe underutilization. For instance, VP should be no larger than $4 \times Lh$ which is 1024 when targeting the LSTMs with the size of Lh being 256, illustrated by the blue line in Fig. 3.7. When EP increases, the number of processing cycles decreases until EP reaches these sweet spots. When EP is larger than the ones in sweet spots, processing cycles increase gradually. Please note $EP = 1$ is not shown in Fig. 3.6(left) as its value is too large and it will make the sweet spot too small to be shown.

From our design space exploration result, the optimal configuration has an EP value between 4 and 16 when $NPE = 16,382$, and between 16 and 64 when $NPE = 65,536$. In these sweet spots, high parallelism can be achieved, which results in high system throughput. Note that for a given vector size, better performance and utilization can be obtained by adapting the (EP, VP) design parameters. As shown in Fig. 3.6 (left), the LSTM workload of 512 has lower latency with (EP, VP) of (32, 2048) than that of (16, 4096) as shown by the red line, while the workload of 1024 has lower latency with (16, 4096) than (32, 2048) as shown by the gray line. Different choices of EP and VP impact the hardware utilization and performance

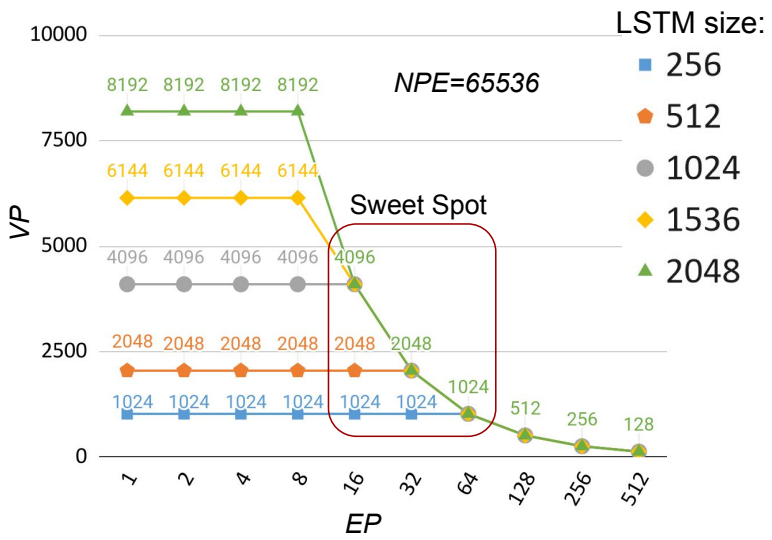


Figure 3.7: Feasible sets of (VP, EP) when NPE equals 65536. The Sweet Spot is set according to the sweet spot in Fig. 3.6(left).

of the architecture when running RNN models of different sizes. There is a trade-off between performance and design complexity with extra hardware resources for supporting various values of (EP, VP). More details about this trade-off is given in the following section.

3.4 Hardware Implementation and Optimization

This section presents our proposed hardware architecture (Fig. 3.8), based on the optimization techniques introduced above. It consists of the kernel units, an adapter unit, an activation function unit and tail units.

3.4.1 Kernel Units

The architecture consists of VP kernel units, and each unit has EP Processing Elements (PEs), so the number of effective PEs is $VP \times EP$. The VP and EP values are determined via the design space exploration described in detail in Section 3.3. In this design, each PE is one fully-pipelined multiplier. Fig. 3.9c shows the details of a computational kernel unit in our design. The architecture of kernel units, as shown in Fig. 3.9a, which employs many

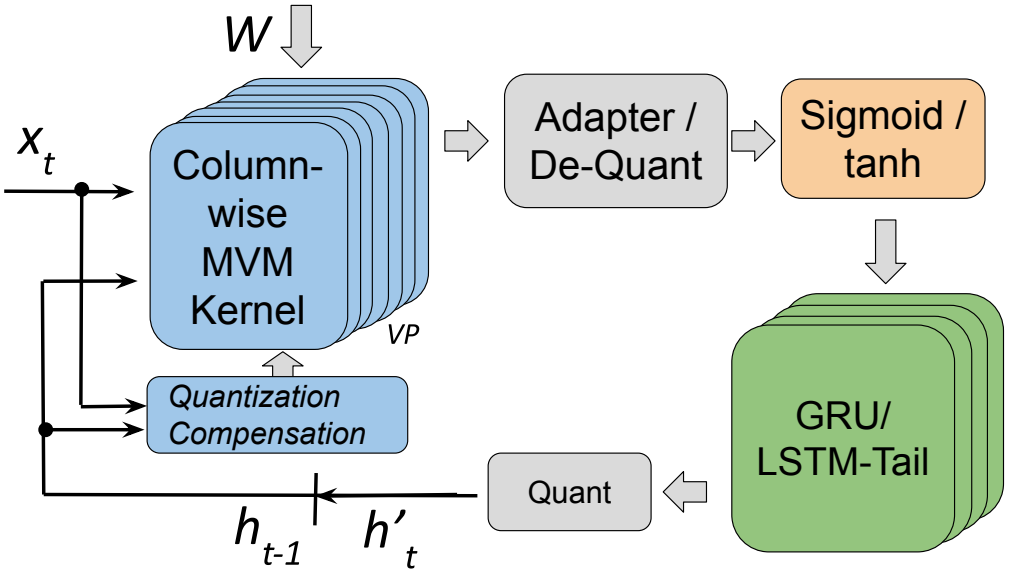
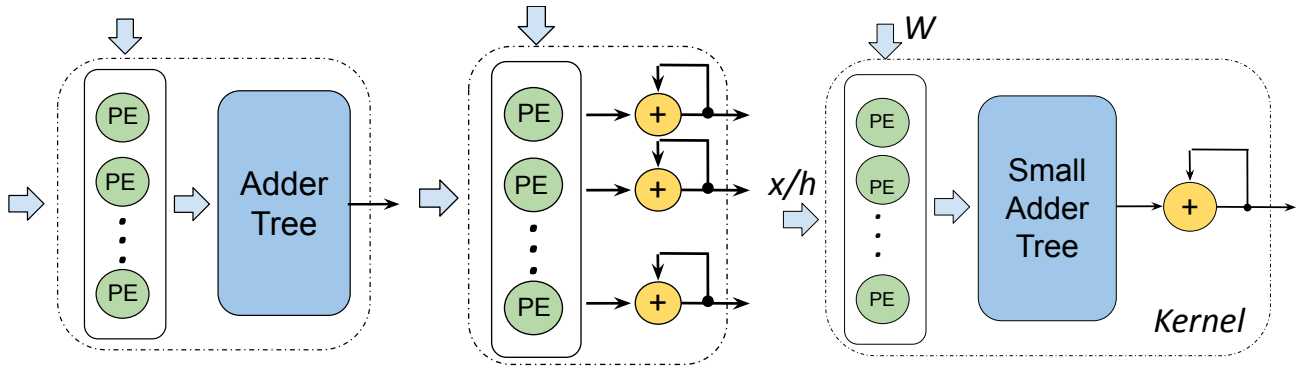


Figure 3.8: The overview of the system

parallel multipliers followed by an balanced adder-tree is commonplace in FPGA-based designs of RNNs [38, 1]. This architecture is for row-wise MVMs. The column-wise MVM is based on the architecture of many parallel multipliers followed by many parallel accumulators, as shown in Fig. 3.9b, since the elements in the partial result vector are not related. To support element-based parallelism, we propose a hybrid hardware architecture that combines these two architectures. A small balanced adder tree is placed between the multipliers and the accumulators, as shown in Fig. 3.9c. This small adder tree, which provides the summation of the products of EP multiplications, can help to balance the EP and VP for a proper shape of a tile. For example, when the VP is limited by $4 \times Lh$ as shown in Fig. 3.7, the design can increase the EP to enable more PEs to increase the throughput since the number of effective PEs is $VP \times EP$.

The hybrid kernel might look more complex than the row-wise or column-wise kernel but it does not consume more hardware resources when targeting a same problem size. For example, if the number of the multipliers is N (for simplicity, N is a power of 2) for all 3 types of kernels and each hybrid kernel has a EP-to-1 small adder-tree, the design with row-wise kernels requires N multipliers and $N-1$ adders (the design is supposed to be a fully pipelined one with a balanced tree), the design with column-wise kernels requires N multipliers and N accumulators, and the



(a) Row-wise kernel (b) Col.-wise kernel (c) Hybrid kernel

Figure 3.9: Various Kernels

design with hybrid kernels needs N multipliers, $\frac{N}{EP} \times (EP-1)$ adders and $\frac{N}{EP}$ accumulators. Because one EP-to-1 balanced adder-tree unit needs $EP-1$ adders and there are $\frac{N}{EP}$ of such units, so there are $\frac{N}{EP} \times (EP-1)$ adders in total. Generally, an accumulator is just an adder so the hybrid kernel also needs $\frac{N}{EP} \times (EP-1) + \frac{N}{EP} = N$ adders. Thus the design with hybrid kernels has the same amount of hardware resources as the one using pure column-wise kernels and just one more adder than the one using row-wise kernels. Please note that some row-wise architectures also have accumulators after the large reduction tree since it can never guarantee to fully unroll the matrix dimension [2]. In such a case, the design using row-wise kernels also requires N adders.

3.4.2 Configurable Adder-tree Tail (CAT) Unit

To support various versions of EP and VP, we design novel adder reduction based on a custom adder-tree with the configurable adder-tree tail. With various EPs, the number of levels of the adder-tree needs to be changed correspondingly. If a fixed structure of the adder-tree is designed for a large EP (EP is also the number of the input elements for the adder-tree), the results from the last several levels of adder-tree can be used for small EPs to update the accumulators directly instead of entering next level adders, as shown in mode 2 and 3 in Fig. 3.10. For example, if EP is 64 and the number of input is also 64, with the proposed 4-input CAT, the number of the output elements of this adder-tree becomes 2 when mode 2 is enabled. Thus, instead of enabling a 64-to-1 adder-tree reduction, the design now has a 64-to-2 adder-tree

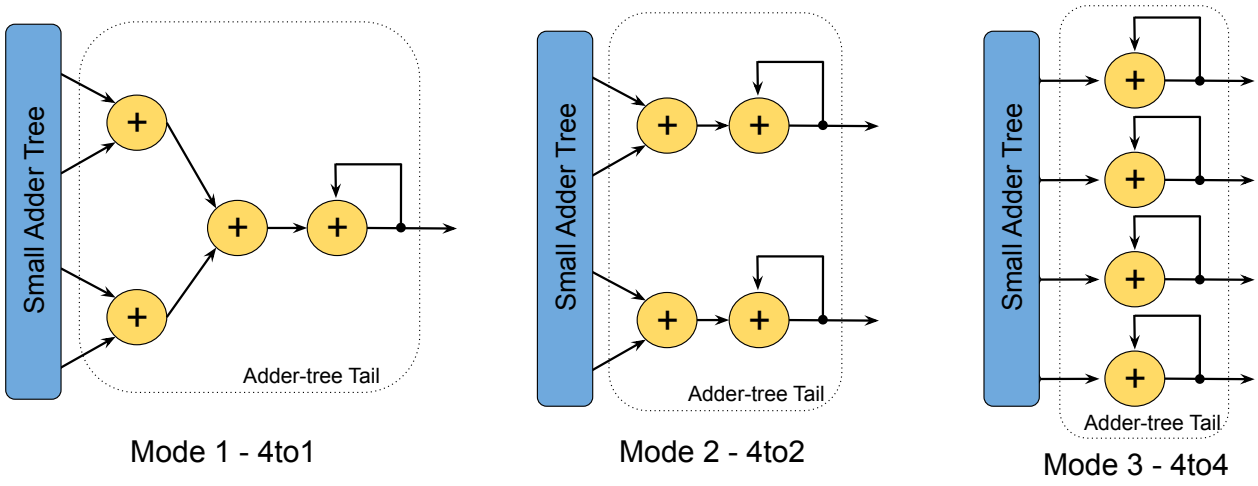


Figure 3.10: The three modes of configurable 4-input adder-tree tail with accumulators

which actually includes two 32-to-1 adder-trees. So the configuration of (EP, VP) changes from (64, 1024) to (32, 2048). With mode 3, the same design can be enabled with (16, 4096). The detailed implementation can be achieved by additional accumulators while keeping the tree structure intact. However, the additional accumulators will result in resource overhead. This work proposes the CAT architecture to reuse the adders in the tail of the tree as the required accumulators with no extra adder components. The CAT with N-input (CAT-N) can be configured to update 1 to N accumulators when the data reach the last $log_2(N)$ levels of the adder-tree. Fig. 3.10 shows the three modes using CAT-4. The results from the adder-tree can be used to update 1 to 4 accumulators. Fig. 3.11 shows the details of a CAT-4 unit. Different MUX settings configure CAT-4 to be one of the 3 modes shown in Fig. 3.10. For example, the red line in Fig. 3.11 shows the data flow when CAT-4 in mode 2. In our large scale design, CAT-4 is sufficient since the sweet spot of EP is {16, 32, 64} according to design exploration for optimal system throughput.

3.4.3 The Other Units

The adapter converts the parallelism between kernels and tails. Then de-quantization (De-Quant) converts quantized values into fixed-point values to reduce hardware resources. The σ/tanh unit performs the Sigmoid (σ) and hyperbolic tangent ($tanh$) functions. Both target

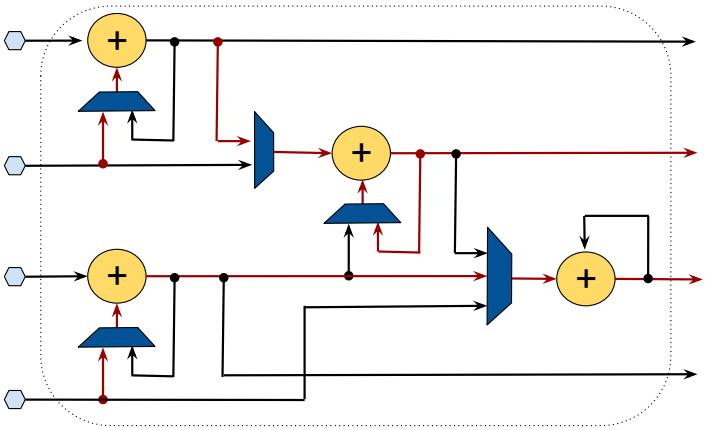


Figure 3.11: The details of CAT-4

programmable lookup tables of size 2048 [9, 2]. The LSTM-tail unit and GRU-tail unit mainly perform the element-wise operations. The output hidden vector (h_t) needs to be quantized before it can be used in the MVM kernels, so a Quant unit is deployed after the final output of Tail units as shown in Fig. 3.8.

3.4.4 Low Precision Multiplications with DSP block Sharing

Reducing the precision of operations in DNN inference accelerators can achieve high efficiency with little or no accuracy loss compared to floating-point by fitting more multipliers per unit area. With careful retraining, low precision, even binarized RNNs can still have decent accuracy [16, 18]. The authors in [16] trained an LSTM model using 1-bit weights and 2-bit activations, which achieved a classification accuracy of 94% for OCR applications. Besides, narrow bit-width multiplications can be mapped efficiently onto lookup tables and DSPs. For example, Brainwave [1] deploys 96,000 MACs on a Stratix 10 2800 FPGA by packing 2-bit or 3-bit multiplications into DSP blocks combined with cell-optimized soft logic multipliers and adders. Our fixed-point 8-bit design has 16,384 MACs. Besides, our fixed-point 2-bit design has 65,536 MACs, and it deploys the same word length for multipliers as those in Brainwave targeting the same FPGA device for a fair comparison. Please note that our column-wise MVM optimizations and fine-grained tiling strategy are applicable to multipliers of any numerical precision.

In current FPGAs, there are highly configurable DSP blocks which are often underutilized in implementing low precision DNN designs. [95] and [96] demonstrated methods to pack two 8-bit multiplications into one Xilinx and Intel 18-bit multiplier respectively. Both methods require two multiplications to share one input operand. With the proposed column-wise MVM, one column of the weights matrix naturally shares the same element of the input vector, which helps us to pack four 8-bit or ten 2-bit multiplications into one DSP block on Intel FPGAs [96] to reduce the hardware resources. Moreover, this would not be a restriction (and will come at lower cost) if we use a novel DSP similar to what was proposed in [97] and will be adopted in the next generation Agilex devices [98].

3.5 Evaluation and Analysis

This section presents evaluation results and analysis of two generations of Intel FPGAs to show the scalability of the proposed RNN accelerator optimizations.

3.5.1 Experimental Setup

To evaluate our RNN design, we utilize the same LSTM and GRU workloads as the Brainwave design [1], the Brainwave-like NPU [2] and the Plasticine [3] for comparison. These workloads come from the DeepBench suite which is a set of micro-benchmarks containing representative layers from popular DNN models such as DeepSpeech [27]. These workloads are single layers with various sizes of hidden vectors and different timesteps (or sequence lengths) from 1 to 1500, as shown in Table 3.3. This work takes the LSTMs with Lh=256 as small LSTM workloads, Lh=512 or 1024 as medium-sized ones, and Lh=1536 or 2048 as large ones. Two generations of Intel FPGAs, an Arria 10 1150 (A10) and Stratix 10 2800 (S10) are evaluated and compared with previous work. Both run persistent LSTM/GRU of inference. Our proposed hardware architecture is captured in Verilog hardware description language, and is implemented in the target A10 and S10 devices using Quartus Pro 18.1.

Table 3.2: Resource Utilization

		ALMs	M20K	DSP	Freq.
Arria 10 1150 (Precision = 8-bit, NPE = 4096)	Avail.	427,200	2713	1518	259Mhz
	Used	186,534	1,178	1,176	
	R. Util.	44%	43%	77%	
Stratix 10 2800 (Precision = 8-bit, NPE = 16,384)	Avail.	933,120	11,721	5760	260Mhz
	Used	487,232	10,061	4,368	
	R. Util.	52%	86%	76%	
Stratix 10 2800 (Precision = 2-bit, NPE = 65,536)	Avail.	933,120	11,721	5760	250Mhz
	Used	768,406	6,803	5,368	
	R. Util.	82%	58%	93%	

3.5.2 Resource Utilization

Table 3.2 shows the resource utilization of our designs with three configurations on FPGAs. We implement a small RENOWN with the configuration of (EP, VP) as (4, 1024) using an Arria 10 FPGA which has 4096 8-bit multipliers in the MVM kernels. A medium-sized RENOWN with the configuration of (EP, VP) as (16, 1024) is implemented using a Stratix 10 FPGA which includes 16,384 8-bit multipliers. A large RENOWN with 65,536 2-bit multipliers is also implemented on a Stratix 10 FPGA with the configurations of (EP, VP) as (16, 4096), (32, 2048) and (64, 1024). All our designs consume most of the FPGAs' available resources. Note that hardware utilization is different from resource utilization and it reflects how often the hardware computational units would be "not idle". Although we achieve a similar frequency to that reported in the Brainwave [1] and Intel-NPU [2] papers, we believe that further low-level optimizations can lead to higher frequencies for better performance. We leave that for future work since it has a limited impact on the conclusions in this work.

3.5.3 Performance and Efficiency Comparison

To illustrate the benefits of our proposed approach, some existing LSTM/GRU accelerator designs using the same benchmark are compared with ours in Table 3.3. This table illustrates the latency, hardware (HW) utilization and throughput with various workloads under different numbers of hidden units (h) and time-step (TS). The hardware utilization is the percentage

of achieved Tera Operations Per Second (TOPS) to the peak performance for each layer. The DeepBench published results [3] on a modern NVIDIA Tesla V100 GPU with 16-bit precision are also included. Some existing FPGA-based LSTM accelerator designs are listed in Table 3.4. For a fair comparison, we only show the previous work with a detailed implementation of the LSTM system in this table. We show the FPGA chips, model storage, precision, number of processing elements (NPE), run-time frequency, throughput, power efficiency and hardware utilization.

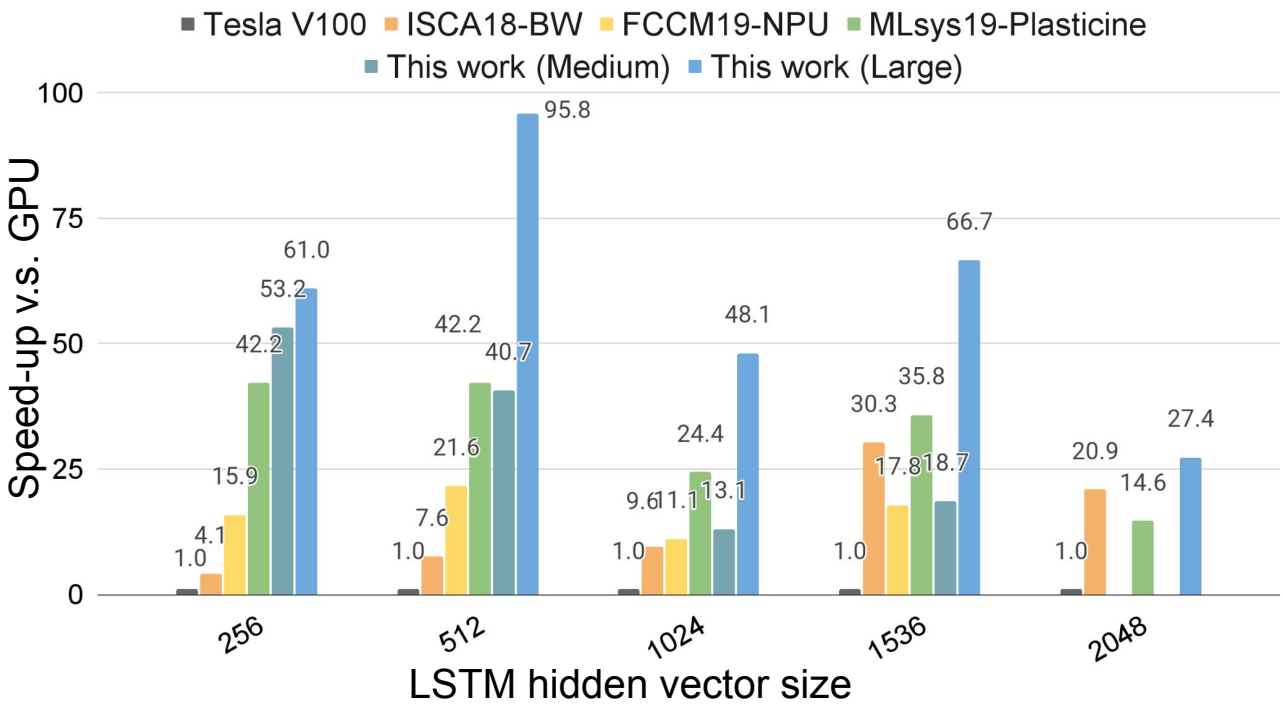
The GPU is significantly underutilized even when cuDNN library API calls are used, since it is designed for throughput-oriented workloads, and it prefers BLAS level-3 (matrix-matrix) operations which are not common in RNN workloads [3]. Our design can provide promising latency under 3ms for all Deepbench RNN layers at batch size of one, reaching up to 29.6 effective TOPS for a large LSTM workload (h=2048) which is the largest reported performance in all these LSTM designs as shown in Table 3.3 and Table 3.4. Our work achieves 27.4 to 95.8 times higher performance than the Tesla V100 as shown in Fig. 3.12a. The performance of the specific case (h=512) is approximately two orders of magnitude higher than the Tesla V100.

Our experiments show that the utilization is low with small RNN applications that are composed of sequences of small MVMs due to small hidden unit sizes and large number of time-steps. However, with our proposed optimizations, we can get higher throughput and hardware utilization than the counterparts using a similar number of PEs. With a similar number of PEs to [2], our RENOWN (Medium-size) achieves up to 94.1% hardware utilization which is the highest with respect to state-of-the-art implementations on FPGAs, as shown in Fig. 3.1 and Fig. 3.12b. Achieving high utilization using a small number of PEs is easier than using a large number of PEs.

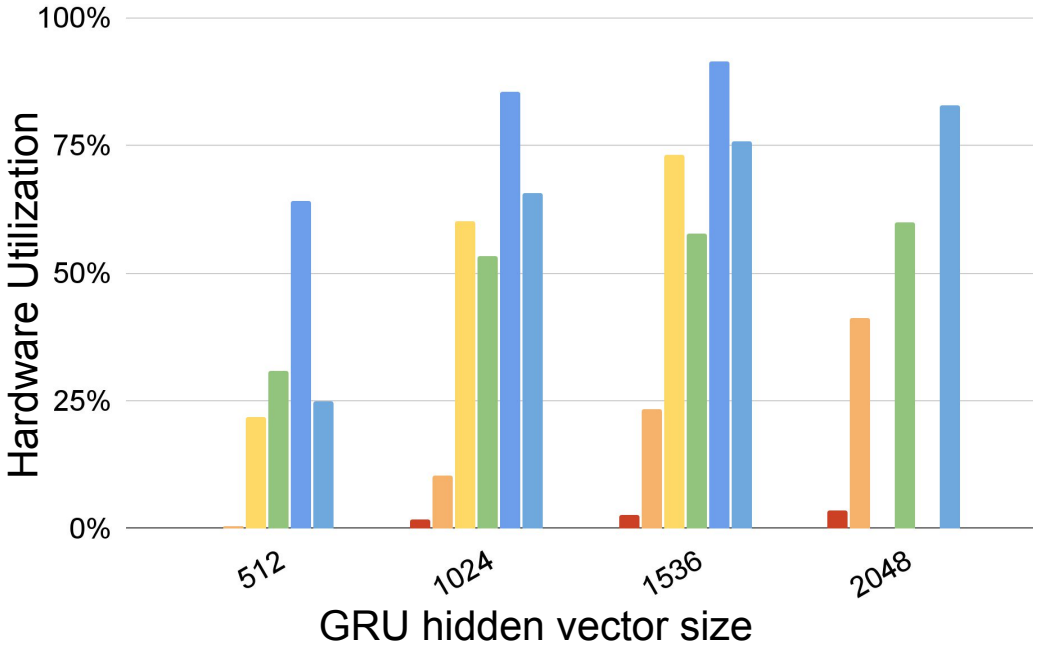
Table 3.3: Performance comparison of DeepBench Inference for the previous work and our designs

Benchmark		GPU	ISCA18-BW[a] [1]	FCCM19-NPU [2]	MLsys19-Plasticine[b] [3]	This Work (Medium)	This Work (Large)
	Process(nm)	12	14	14	28	14	14
	NPE	-	96000	19200	12288	16384	65536
GRU h=512 TS=1	Latn. (ms)	0.39	0.013	0.0015	0.0004	0.0006	0.0004
	HW Util.	0.03%	0.5%	21.7%	30.9%	64.1%	24.8%
	Perf. (TOPS)	0.01	0.25	2.17	7.6	5.46	8.13
GRU h=1024 TS=1500	Latn. (ms)	33.77	3.792	3.139	1.4430	2.59	0.879
	HW Util.	1.8%	10.4%	60.2%	53.3%	85.5%	65.5%
	Perf. (TOPS)	0.56	4.98	6.01	13.1	7.28	21.5
GRU h=1536 TS=375	Latn. (ms)	13.12	0.951	1.454	0.7463	1.36	0.428
	HW Util.	2.6%	23.3%	73.2%	57.8%	91.4%	75.8%
	Perf. (TOPS)	0.81	11.17	7.30	14.2	7.79	24.8
GRU h=2048 TS=375	Latn. (ms)	17.70	0.954	-	1.283	-	0.695
	HW Util.	3.4%	41.2%	-	59.81%	-	82.8%
	Perf. (TOPS)	1.07	19.79	-	14.7	-	27.1
GRU h=2560 TS=375	Latn. (ms)	23.57	0.993	-	1.973	-	1.076
	HW Util.	4.0%	61.8%	-	61.0%	-	83.6%
	Perf. (TOPS)	1.25	29.69	-	15.0	-	27.4
LSTM h=256 TS=150	Latn. (ms)	1.69	0.425	0.110	0.0419	0.033	0.029
	HW Util.	0.3%	0.8%	14.3%	15.5%	56.1%	16.7%
	Perf. (TOPS)	0.09	0.37	1.43	3.8	4.79	5.49
LSTM h=512 TS=25	Latn. (ms)	0.60	0.077	0.027	0.0139	0.014	0.0061
	HW Util.	0.6%	2.8%	38.8%	30.9%	85.9%	52.6%
	Perf. (TOPS)	0.18	1.37	3.89	7.6	7.33	17.2
LSTM h=1024 TS=25	Latn. (ms)	0.71	0.074	0.064	0.0292	0.054	0.015
	HW Util.	1.9%	2.8%	65.7%	58.6%	90.7%	86.6%
	Perf. (TOPS)	0.59	5.68	6.56	14.4	7.73	28.4
LSTM h=1536 TS=50	Latn. (ms)	4.38	0.145	0.246	0.1224	0.236	0.066
	HW Util.	1.4%	27.1%	76.9%	63.7%	94.1%	87.4%
	Perf. (TOPS)	0.43	13.01	7.67	15.4	8.02	28.7
LSTM h=2048 TS=25	Latn. (ms)	1.55	0.074	-	0.106	-	0.113
	HW Util.	3.4%	47.1%	-	64.3%	-	90.2%
	Perf. (TOPS)	1.08	22.62	-	15.8	-	29.6

[a] Brainwave, [b] ASIC.



(a) Performance speedup for the various LSTM dimensions



(b) HW utilization of various GRUs compared to prior work

Figure 3.12: Performance comparison

Table 3.4: Comparison with previous implementations of LSTM on FPGAs

	FPGA17 ESE [9]	FPL18- [15]	ISCA18- BW [1]	FCCM19- NPU [2]	FCCM19- NPU [2]	JSPS20- [16]	FPGA20- [56]	TVLSI20- [99]	This work (Medium)	This work (Large)
FPGA chips	Kintex KU060	Zynq ZU7EV	Stratix10 GX2800	Stratix10 GX2800	Stratix10 GX2800	Zynq ZU9EG	Zynq ZU9EG	Zynq ZU6EG	Stratix10 GX2800	Stratix10 GX2800
Process (nm)	20	16	14	14	14	16	16	16	14	14
Model Storage	-	on-chip	on-chip	on-chip	on-chip	on-chip	on-chip	on-chip	on-chip	on-chip
Precision (bits)	12 fixed	1-8 fixed	BFP-1s5e2m	4 fixed	8 fixed	2 fixed	4/8 fixed	16 fixed	8 fixed	2 fixed
DSP Avail.	2,760	1,728	5,760	5,760	5,760	2,520	2,520	1,973	5,760	5,760
DSP Used	1,504	-	5,245	4,880	5,600	6	64	-	4,368	5,368
NPE	1024	-	96,000	19,200	38,400	-	-	-	16,384	65,536
Freq. (MHz)	200	240	250	255	260	240	240	385	260	250
Power (W)	41	-	125	65.3	67.6	15.5	15.5	0.95	62	98
Perf. (GOPS)	282	1,833	370[a] 22,620	- 14,112	1,431[a] 7,980	3,618	3,072	18	4,790[a] 8,015	5,489[a] 29,574
Power Effi. (GOPS/W)	6.87	-	180	216	118	234	199	19	129	302
LSTM HW Utilization	-	-	0.8%[a] 47.1%	-	14.3%[a] 76.9%	-	-	-	56.1%[a] 94.1%	16.7%[a] 90.2%

[a] When targeting a small LSTM model (Lh=256).

In our RENOWN (Large) design, we use around 31.7% fewer multipliers than [1], while achieving 30.7% higher performance than [1] when targeting large LSTM workloads. When targeting small LSTM models, our design achieves 14.8 times higher performance. This is due to better hardware utilization which comes from our optimizations incorporating novel column-wise MVM and fine-grained tiling strategy. With a large number of PEs, our RENOWN can still achieve up to 90.2% hardware utilization which is higher than the other work. When targeting small LSTMs and GRUs, RENOWN (large) has a similar utilization as [2] and [3]. However, the number of PEs is respectively 3.41 times and 5.3 times more than those in [2] and [3]. The configurable tiling of RENOWN (Large) also results in an additional up to 27% higher system throughput as shown in Fig. 3.13. The figure shows the speedup of the system throughput compared to the lowest one among them with a given EP value.

The previous designs of [16] and [56] in Table 3.4 explore various low precision designs, showing their scalability to different bitwidth designs. Most of the other designs in Table 3.4 only support one bitwidth while our work demonstrates both 8-bit and 2-bit designs which show the scalability of our architecture to cover different bit-width designs. Our implementations are produced using Verilog templates with configurable parameters for each hardware module instance. Generally, [16] and [56] are more scalable in bitwidth since they do not use the DSP hard blocks on FPGAs for the computational kernels. The FPGA DSP block has a fixed bit-width and it needs a tailor-made wrapper for packaging several small multipliers into one DSP hardware block. Besides, [16] and [56] target a particular model which has four parallel and independent LSTM layers. These four independent LSTM layers can be scheduled in an interleave manner for a hardware engine to alleviate the issues of recurrent dependencies.

Some of the designs target smaller FPGAs [16, 56] than the Stratix 10. They have fewer logic resources and DSPs but they also have a smaller TDP (thermal design power) power consumption. We follow the convention in all related papers and use GOPS for performance comparison. In addition, we also use GOPS/W, power efficiency, so that we are not taking advantage of the FPGA chip size when compared to the designs on small chips, such as the designs in [16, 56].

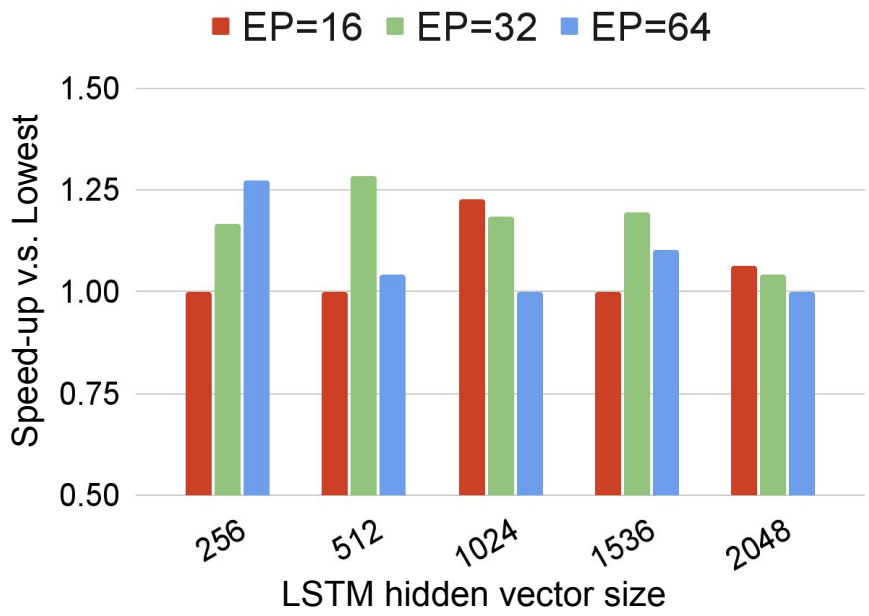


Figure 3.13: Performance speedup due to configurable tiling

Overall, our RENOWN (Medium-size) provides over 1.05 to 3.35 times higher performance and 1.22 to 3.92 times higher hardware utilization than the state-of-the-art design [2], as shown in Table 3.3. In addition, the RENOWN (Large) achieves 3.7 to 14.8 times better performance than state-of-the-art FPGA-based LSTM designs [1, 2, 3]. This work focuses on minimizing latency and maximizing throughput by increasing the hardware utilization. The results show flexible customizability of our architecture for different scenarios. The column-wise approach exposes the most parallelism while minimizing stalls due to data dependencies.

To minimize latency, our design places the model weights onto the on-chip memory, achieving high memory read bandwidth suitable for real-time services. However, some large-scale RNNs recently emerge with large on-chip memory requirements. Our design adopts a scale-out network of utilizing multiple accelerators like [1, 66] which partition the design to multiple FPGAs to address the challenges of fast-growing RNN models where weights exceed on-chip memory capacity on a single FPGA. Some cloud-based services are able to tolerate a slightly longer latency of response. It means a small amount of batching can be employed if necessary. This benefits the GPU-based RNN inferences [1]. In [18, 15, 21, 22], the batching technique is used to improve the hardware throughput and utilization for LSTM inferences. Since our design consumes a single input at a time, increasing batch size does not affect its utilization. Thus,

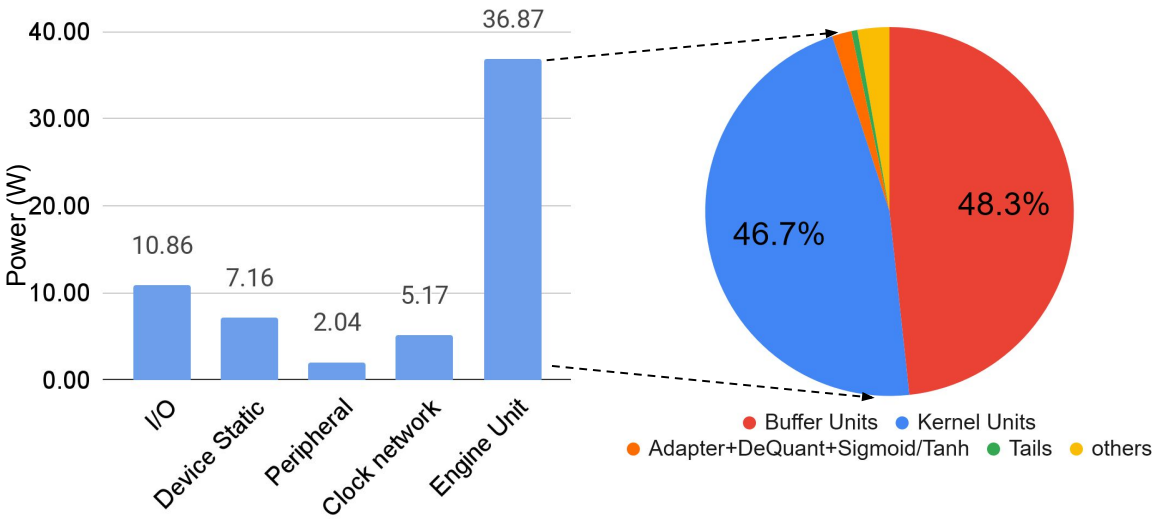


Figure 3.14: Power consumption of the accelerator with the decomposition for each of the major underlying blocks.

our architecture's utilization is not affected with or without batching. Furthermore, some designs use binarized datapaths [15, 16, 56] for LSTMs with negligible or no effect on accuracy. Utilizing very low precision, e.g., binary, is orthogonal to our proposed approach which transforms computation to eliminate data dependencies. Reducing precision can be combined with our approach to achieve even higher performance and efficiency.

The comparisons made in this work do come with certain limitations. One key limitation is that the impact of varying bitwidths on computational accuracy is not directly assessed in our comparisons. Although reducing bitwidth can enhance efficiency, it's important to acknowledge that this change can also influence accuracy outcomes. However, we often lack access to the detailed accuracy information of the existing studies we're comparing. Besides, the benchmark used does not fully represent real-world end-to-end application scenarios. Instead, it focuses primarily on typical RNN layers which does not inherently include model accuracy information. Furthermore, the accuracy/throughput in application level can be fairer metrics. However, it is complicated as the listed previous papers utilize various applications to benchmark their designs. We decide to use GOP/S as our comparative metric since it is commonly used in relevant literature.

Fig. 3.14 illustrates the power consumption of the proposed RENOWN (Medium-size) design.

The power is estimated by the Intel Early Power Estimator (EPE) tool and verified by the Intel Quartus Power Analyzer. We note that this work considers only the chip power, for a fair comparison. The bar chart in Fig. 3.14(left) shows the decomposition for each of the major FPGA components. The static power consumption of the device is 7.16W. The dynamic power consumption of the accelerator engine unit is the largest which is over 50%. The pie chart in Fig. 3.14(right) shows the dynamic power consumption of each major unit of the accelerator engine unit. The Buffer Units which store the weights and input/output data have the largest power consumption which reaches nearly half of the power consumed by the whole engine unit. The Kernel Units also consume nearly half of the power while the Tail units only consume less than 1% power.

3.6 Summary

This chapter presents a novel column-wise MVM for RNNs to eliminate data dependencies and introduces a latency-hiding hardware architecture with hybrid kernels and Configurable Adder-tree Tail (CAT) units, increasing the hardware utilization and system throughput. It also introduces a flexible checkerboard tiling strategy that supports EP and VP to exploit the available parallelism while increasing hardware utilization and scalability. The proposed accelerator has been implemented using Arria 10 and Stratix 10 FPGAs, achieving superior performance and power efficiency compared to prior state-of-the-art implementations.

Chapter 4

Optimizing Large RNNs with Weight Reuse

4.1 Introduction

This chapter introduces a weight reuse approach for large RNN models with weights stored in off-chip memory, running with a batch size of one. A novel blocking-batching strategy is proposed to optimize the throughput of large RNN designs on FPGAs by reusing the RNN weights. It is important to note that our strategy diverges from traditional batching techniques that group completely different input requests to exploit the inter-request parallelism at the cost of latency, as discussed in Section 1.1.2. Instead, our blocking-batching approach involves the batching of activations of different timesteps from the same input sequence.

Although FPGA-based LSTM accelerators have advantages in latency and power consumption, they are limited by the memory bandwidth of the FPGA board. The situation is even worse when we consider a small embedded system with low power and low memory bandwidth. An example of this is a monitoring camera system performing video processing, where large machine learning models have previously been infeasible due to high memory bandwidth and low latency requirements.

There has been previous work [1, 14, 15, 100] with FPGA based implementations such that all the weights are stored in the on-chip memory, but this is expensive and limits the size of models that can be deployed. When the RNN model is so large that the weights need to be stored on an external DRAM, it is not efficient because the fetched weights are typically used only once for each output computation.

In this work, we focus on LSTM models which are too large to store in on-chip memory of FPGA and we propose a novel blocking-batching strategy splitting the weight matrix into multiple blocks while batching the input activation vectors so that we can process the calculations block by block with weight reuse, which will reduce external memory access to save power and reduce latency. Batching the input activation vectors for RNN has been studied [15, 17, 18, 19] to increase the throughput, however few concern combining the blocking and batching for RNNs. In addition, we analyze the underlying data access pattern and dependency in the matrix-vector multiplication required by LSTM and a stall-free hardware architecture is proposed. With our method and new hardware architecture, large LSTM systems can be processed efficiently on FPGAs.

Our contributions are as follows:

1. A new blocking-batching strategy to reuse the LSTM weights to optimize the throughput of large LSTM systems on FPGAs with a stall-free hardware architecture, resulting in high throughput.

2. A performance model which enables a balance between performance, power consumption and area for FPGA designs. Promising power efficiency improvement has been achieved due to less off-chip access.

4.2 Design and Optimization

Since most of the calculations within LSTM cells lie in the matrix-vector multiplication with complex data dependencies, this work will mainly focus on optimizing this operation for high

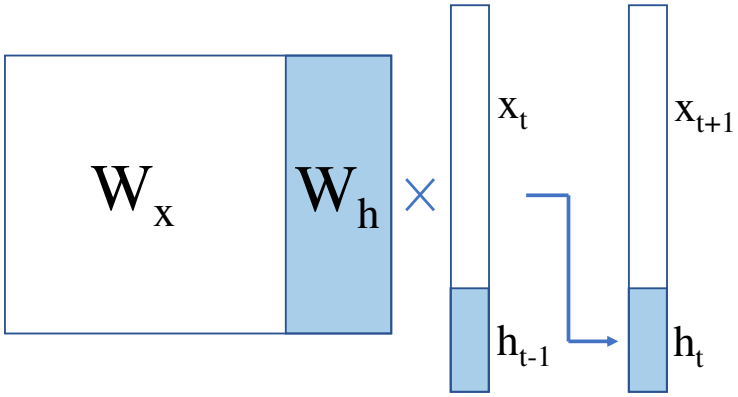


Figure 4.1: Matrix-vector multiplication, showing the data dependency.

throughput. The element-wise operations in the LSTM tail can run parallel with the matrix-vector multiplications. In this section, an improved architecture is first presented to re-organize the multiplications to optimize data dependency and reduce stalls, thereby increasing the throughput of the system. Then a new blocking-batching strategy of reusing the LSTM weights to enhance the throughput of large LSTM systems is described via 3 scenarios. In addition, an approach to optimize the design setting such as finding the best blocking number and the best batch sizes is introduced.

4.2.1 Overcoming Data Dependency

The traditional implementation of the matrix-vector multiplication involves the entire vector of (x_t, h_{t-1}) and a whole row of the weights at a time. However, this approach imposes additional stalling as the system needs to wait for new computed hidden units vector before starting the next time-step. This is mainly due to the data dependency between the output from the current time-step and the vector for the next time-step as shown in Fig. 4.1, where W_x and W_h represent the weights for the input vector and the weights for the hidden vector respectively. That implies that the whole system pipeline needs to be emptied to get the new computed hidden units before the new matrix-vector operations can start.

We propose a new technique that can alleviate this problem by calculating the matrix-vector operations in a different manner. At the beginning, only a few elements from the x_t vector

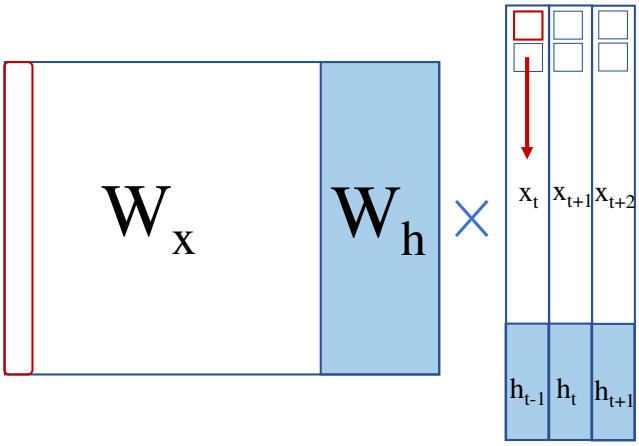


Figure 4.2: New matrix-vector multiplication method using columns.

are used while h_{t-1} is not touched, but all the elements in the corresponding columns of the weights matrix are used to do the operations, as shown in Fig. 4.2. The number of the involved elements in the x_t vector each cycle depends on the parallelism of system. In this way, the calculation of the new inference of (x_{t+1}, h_t) can start without waiting for the system pipeline to be emptied to get the h_t, which means that the system can be fully pipelined without stall. Each hidden vector can finish the computation in the shadow of processing x_t before it is used.

4.2.2 New Blocking-Batching Strategy

Many LSTM designs on FPGA share the same problem where all the weights need to be stored on-chip because of the slow latency to the off-chip memory. This approach is inapplicable for large machine learning models or a small FPGA. Even after model compression and weights pruning, the designs can still suffer from insufficient on-chip memory for large compressed model. To solve this problem, we propose splitting the weight matrix into multiple blocks while batching the input activations vectors so that we can process the calculations block by block with weight reuse. This technique can be used for general LSTM model, or incorporated with the technique proposed in Section 4.2.1. It seems similar to classic block matrix-matrix multiplications, however our proposal also considers the data dependence in LSTMs. With multiple vectors organized in a batch, the system can now reuse the same weights for the next matrix-vector operation without the full input vector being ready. Since memory accesses are

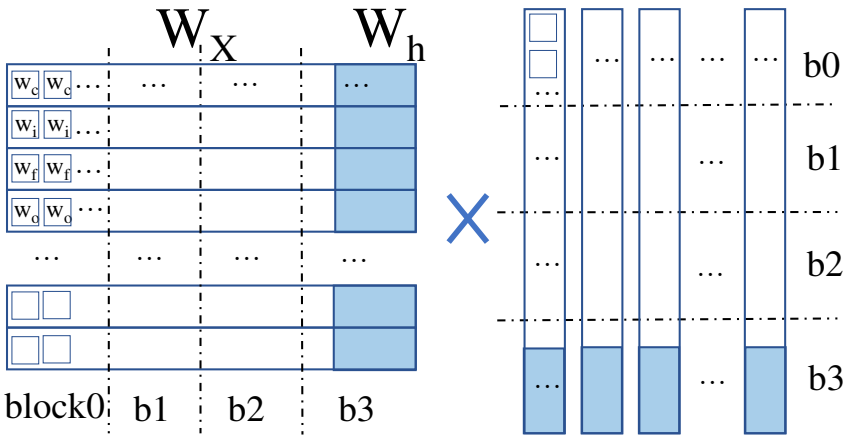


Figure 4.3: Blocking of the weights matrix and input activation vectors.

expensive we want to reduce the number of loads from memory. This method can reuse the weights on multiple input vectors before reloading new weights from memory. This approach is especially useful in an embedded system where FPGA size and memory resources are both limited.

In our approach, the matrix includes parameters from all kinds of LSTM gates. In addition, the gate weights are interlaced in the matrix. Furthermore, we slice the weights matrix along the column, so the number of columns in each block is 1/(Blocking_Number of the original number of columns in the weights matrix), while the number of rows is the same as the original number of rows, as shown in Fig.4.3. So each block includes parameters from all kinds of gates.

Typically the transfer time of the weights is much larger than the computation time. By processing multiple time-steps of the input vector in a batch, we can use the weights multiple times before reloading, which reduces the number of memory accesses. Assuming the number of processing elements is fixed, increasing the batch size will also increase the computation time. We can find a batch size such that the computation time is equal to or larger than the transfer time to hide memory latency. In this way, we convert memory-bound applications to compute-bound ones and improve the performance.

In addition, we make use of a double buffering architecture which stores two blocks on-chip. Whilst calculating one block we can transfer the other block to maximize efficiency by reducing

Table 4.1: Blocking-Batching Parameters

M_{op}	Number of matrix operations
N_{pe}	Number of processing elements
N_t	Number of elements transferred each cycle
N_b	Number of blocks
L_x, L_h	Number of elements in x and h vectors respectively
α	$L_x/(L_x + L_h)$
B	Batch size
DW	Data Width
P [1]	$Performance/(2 \times frequency)$

[1] Performance is in terms of throughput while 2 means each data need both multiplication and accumulation operations.

the stalling time.

4.2.3 Performance Model (Technology Independent)

This subsection presents the performance model which is independent of the particular FPGA technology. There are 3 cases in this blocking-batching strategy:

1. The hidden unit weights can be stored in one block
2. The hidden unit weights can be stored in two blocks
3. The hidden unit weights need be stored in more than two blocks

We define a few parameters, as shown in Table 4.1 for later calculations. Ideally we would like the calculation time for each block to be equal to the transfer time, but in reality usually one is significantly longer than the other. Let us assume the calculation time for one block is longer than the transfer time for one block.

$$\text{Calculation Time} \geq \text{Transfer Time}$$

$$\frac{M_{op}B}{N_b N_{pe}} \geq \frac{M_{op}}{N_b N_t} \implies B \geq \frac{N_{pe}}{N_t} \tag{4.1}$$

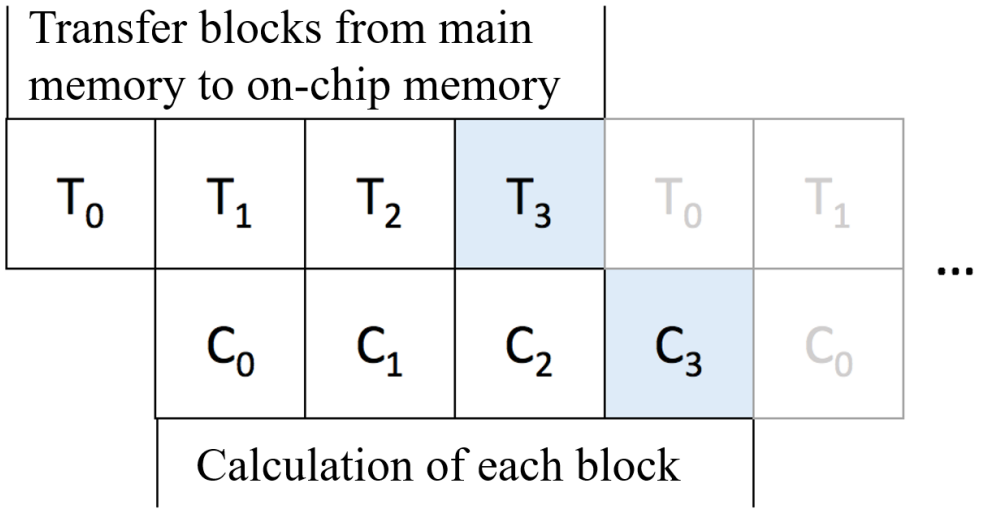


Figure 4.4: Timing diagram for case 1.

This gives us the constraint $B \geq \frac{N_{pe}}{N_t}$ when the calculation time is greater than or equal to the transfer time. Similarly we can derive the constraint $B < \frac{N_{pe}}{N_t}$ when calculation time is less than the transfer time.

Case 1 — In this case, the performance is almost dictated by having to store all weights in the on-chip memory. If the maximum performance without stall is P_m, then this case can achieve P_m. This is due to the novel stall-free blocking-batching architecture that ensures we are always calculating and there is no stalling.

The ideal timing diagram for this case is shown in Fig. 4.4, where there is no idle time. T0 is transfer time for Block0 while C0 is computation time for Block0. As shown in Fig.4.4, C0 can start when T0 has finished. In practice, we find that there are some special cases where we must stall the pipeline to wait for the final block to finish calculating. Normally we can ignore the system latency because we can start processing the x part of the final block before we reach the h elements, as illustrated in Fig. 4.2; by the time we reach the h elements they will be ready. If the hidden input vector h occupies a large amount of the block, then we will have to wait for the system pipeline to finish processing the last vector, which will cause stalls. We find that these stalls cause the calculation of the final block to take about 10% longer time. The calculations below consider the simple case when there are no stalls.

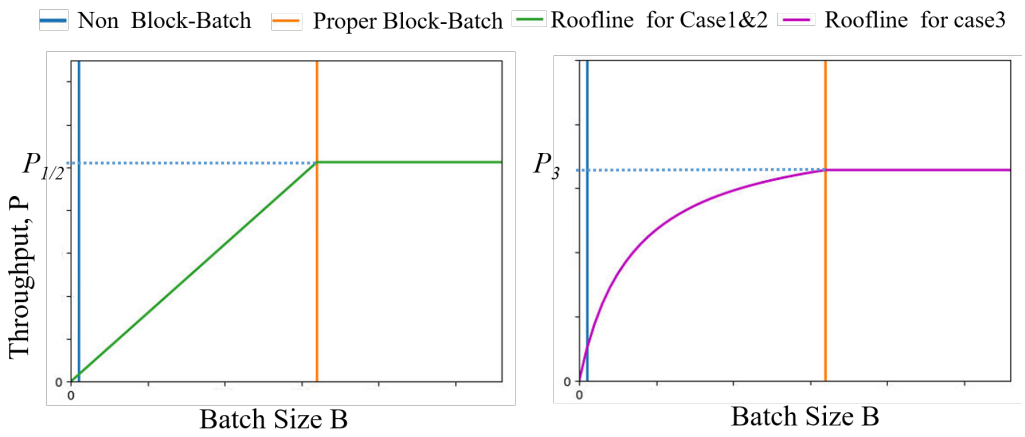


Figure 4.5: Roofline performance model for case 1&2 (left) and case 3 (right).

We can calculate the effect on performance by considering the total number of operations that must be done against the time spent. The performance depends on the time we spend transferring each block versus the calculation time of each block, as shown in the following equations and Fig. 4.5.

$$P = \frac{M_{op}B}{\frac{M_{op}B}{N_{pe}}} = N_{pe} \qquad \text{when } B \geq \frac{N_{pe}}{N_t} \tag{4.2}$$

$$P = \frac{M_{op}B}{\frac{M_{op}}{N_t}} = BN_t \qquad \text{when } B < \frac{N_{pe}}{N_t} \tag{4.3}$$

The blocking number, N_b, can be increased to reduce the on-chip memory needed. Due to storing two blocks on-chip, we only need an amount of memory given by $\frac{2}{N_b}$ to store all weights on-chip. This means we can process a model many times larger, or process the same model using a fraction of the on-chip memory. Of course there are some drawbacks to increasing the block number, which are covered in cases 2 and 3.

Case 2 — In this case we must wait for both of the last blocks to be in the on-chip memory before starting computation, because the next hidden vector in the batch has a dependency on the previous. Fig. 4.6 shows the timing diagram for this case, where the red arrows indicate

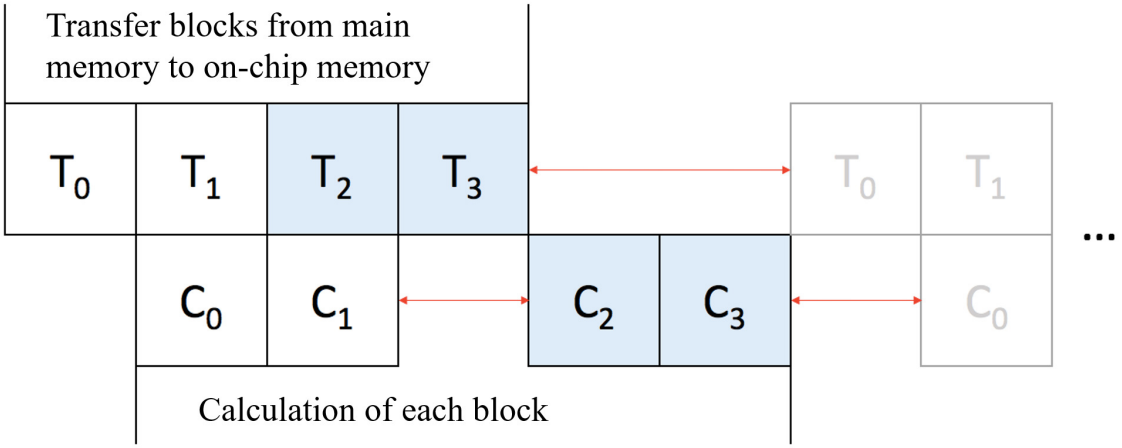


Figure 4.6: Timing diagram for case 2. The red arrows indicate the extra time we must wait, i.e. stall.

the extra time we must wait.

$$P = \frac{M_{op}B}{\frac{M_{op}B}{N_{pe}} + \frac{2M_{op}B}{N_b N_{pe}}} = \frac{N_{pe}N_b}{N_b + 2} \tag{4.4}$$

In theory there is a small overlap at the beginning where we can begin to compute the first sub-vector, and also at the end when we can start transferring while working on the last sub-vector in the last vector of the batch. Since this is equal to double the *time to process one sub-vector*, it will be negligible compared to the total time and we shall leave this out of our approximations. The performance calculation is done in a similar way when $B = \frac{N_{pe}}{N_t}$; we consider that each matrix element must be transferred and the transfer time is equal to time for processing all the M_{op}, but the hidden weights also have the added processing time which takes up 2 blocks in all N_b blocks.

In cases 1 and 2, we can achieve weight reuse for both the independent parts of the vector (input activations) and the dependent components (hidden vector) within the batch. By utilizing weight reuse, we can decrease external memory traffic down to a fraction of $\frac{1}{B}$. It means that the design only loads the weights once and then efficiently reuses them across all vectors in the batch, thereby reducing external memory access, leading to improved design efficiency.

Case 3 — In the most complex case we have multiple blocks due to a large LSTM model

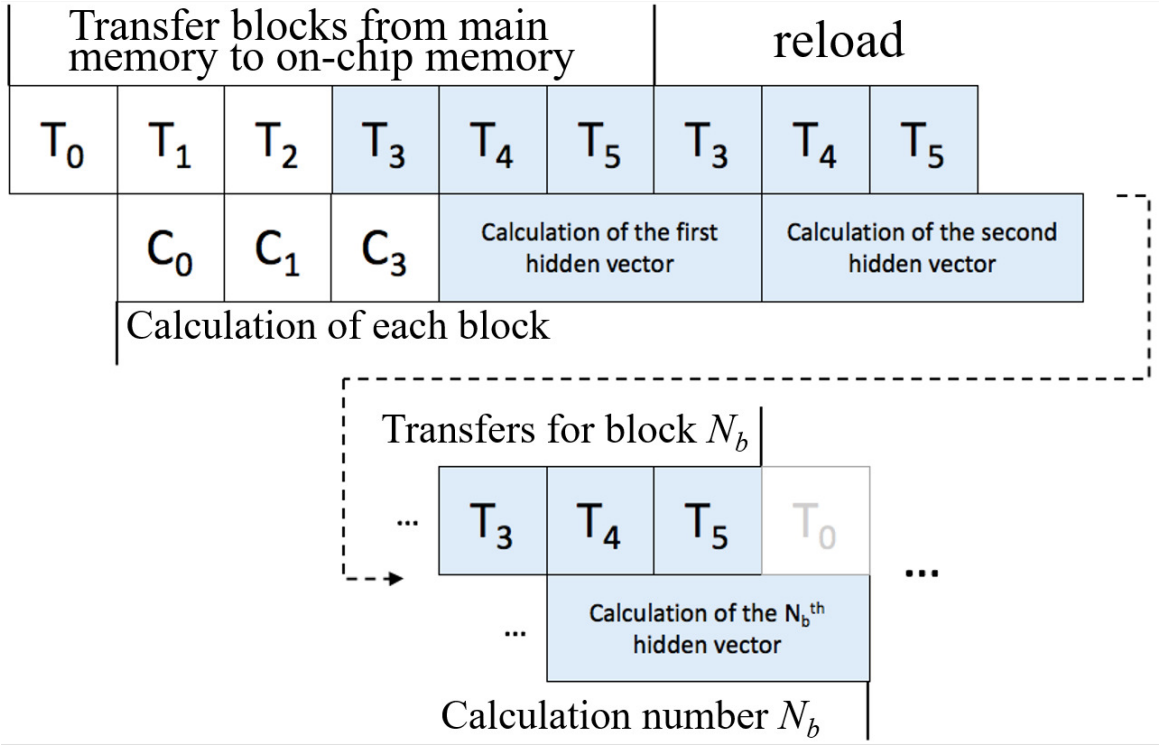


Figure 4.7: Timing diagram for case 3.

and/or a small FPGA. In this case the hidden vector will be split across more than two blocks, so we cannot store it entirely on-chip at the same time.

Due to the data dependency between sub-vectors we must reload the last few blocks where the hidden vector is. It is necessary to reload N_b number of times to finish each vector in the batch.

The performance calculation is more complex but follows the same pattern as before. We consider each case, when the x_t input and weights takes longer to transfer, then $\frac{\alpha M_{op}}{N_t}$ is larger than $\frac{\alpha M_{op} B}{N_{pe}}$ as shown in equation (4.6), or when the calculation takes longer than $\frac{\alpha M_{op} B}{N_{pe}}$ as shown in equation (4.5). Conversely, the hidden input and weights always spend more time transferring since each calculation is only one sub-vector from the batch, yet all the weights need to be transferred each time. The final roofline model is shown in Fig. 4.5.

$$p = \frac{M_{op} B}{\frac{\alpha M_{op} B}{N_{pe}} + \frac{(1-\alpha) M_{op} B}{N_t}} = \frac{N_{pe} N_t}{\alpha N_t + (1-\alpha) N_{pe}} \qquad \text{when } B \geq \frac{N_{pe}}{N_t} \tag{4.5}$$

$$p = \frac{M_{op} B}{\frac{\alpha M_{op}}{N_t} + \frac{(1-\alpha) M_{op} B}{N_t}} = \frac{B N_t}{\alpha + (1-\alpha) B} \qquad \text{when } B \leq \frac{N_{pe}}{N_t} \tag{4.6}$$

Although this case seems to offer poor performance because of the limitation of memory bandwidth, we should remember that this is similar to the standard method without processing using columns. We would need to load each block into memory B times and each sub-vector would be processed individually. With our new architecture, we re-use the weights as much as possible for the independent part of the vector, and only need to reload the weights for the dependent part of the vector with the hidden weights. In this scenario, the memory traffic can be reduced to a fraction of $\frac{B+1}{2B}$. When B is much larger than 1, the r Furthermore, if there are more on-chip memory on the target FPGA then this case will be changed to case 1 which becomes compute-bound with high performance.

4.2.4 Resource Modelling

FPGA-based LSTM accelerators are constrained by two types of resources: one is the logic resources such as LUTs and DSPs, the other is the memory resource i.e. the BRAMs. Based on [101, 102], DSPs are the limiting resource for the computation engines. Therefore, only DSP usage is considered in this category. In our design, fixed-point adders are implemented using LUTs in order to save DSPs since the adders consume much fewer LUTs compared to that of multipliers and considering the available LUTs are far more than the DSPs on FPGA. Let D_{mul} represents the number of DSP usage of one multiplier, the total number of required DSPs is $N_{pe} \times D_{mul}$.

The memory resources are mainly occupied by the dual buffers and BB-FIFOs and its usage is given by:

$$BRAM_{Num} = \frac{(L_x + L_h) \times (4L_h + B) \times DW \times 2/N_b + B \times N_{pe} \times DW}{BRAM_{size}} \tag{4.7}$$

Practically, Blocking-Batching(BB)-FIFO can be implemented using LUTRAM in order to save BRAMs when the entry of each BB-FIFO is small. If so, the term of $\frac{B \times N_{pe} \times DW}{BRAM_{size}}$ in equation (4.7) can be omitted.

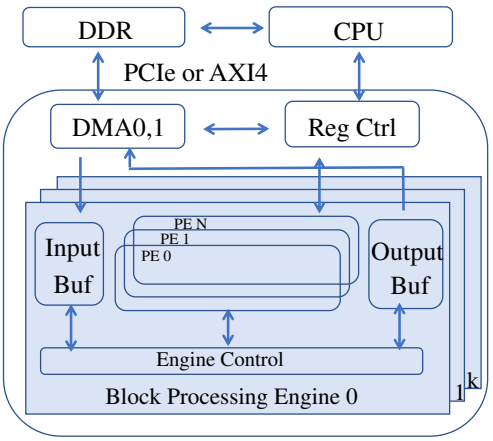


Figure 4.8: The entire system.

4.3 System Architecture

4.3.1 System Overview

Fig. 4.8 shows the overall system on a FPGA board while Fig. 4.9 shows the architecture of the Stall-free Blocking-batching Engine (SBE). This system consists of SBE units, with a CPU and DDR3 DRAM as the off-chip memory. All the weights and input activations (CNN-extracted features) are stored in the off-chip memory. The Reg Ctrl unit, which is connected to the AXI4-lite bus, is used to transfer the control commands while data communication is managed by the DMA units which are connected to the PCIe bus or AXI4 bus. The CPU is used to send configurable parameters to the SBE and control the transmitting of the weights and receiving the results when the hardware finishes processing, which is all done via the Reg Ctrl unit.

The details of the SBE architecture is shown in Fig. 4.9. As mentioned, only one block is transferred from the off-chip memory to the FPGA on-chip memory in each iteration of computation. The partial weights will be stored in buffer0 and buffer1 which work as a double buffer. In addition the partial batch_size activations of the input x vectors are also stored in a double buffer. With a carefully chosen batch size, these buffers work to overlap the time of data communication with LSTM inference computation.

The processing elements (PE) perform the matrix-vector operations that work as the LSTM gates. They multiply one element from the partial input vector by all the corresponding weights.

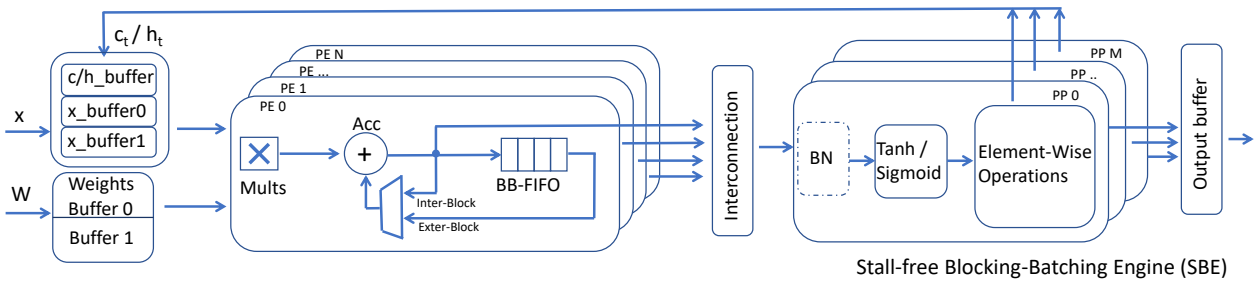


Figure 4.9: Stall-free Blocking-batching Engine (SBE) Architecture Details. The details of LSTMs can be found in the background Section 2.1.2 and Fig. 2.2.

The partial result of one partial activation will be accumulated via the inter-block linking and finally stored into the small Blocking-Batching(BB)-FIFO to be used in the next block. Each partial activation in the batch will generate one result and will be stored in the BB-FIFO. Therefore, the depth of the BB-FIFO is equal to the batch_size.

Consolidating of the block computations is done via the BB-FIFO in the PE units. When the new block computation begins, the value in the BB-FIFO will be read via the Exter-Block link and used as the initial value for the accumulator. After the new block computation, the partial results of the new block will be accumulated into the former partial results and finally stored into the BB-FIFOs. When all the blocks are processed, the final result across all blocks for the batch will be generated on the LSTM interconnection unit, where they will be reshaped for later processing.

4.3.2 SBE Architecture

The post processing (PP) units are used to perform other functions after the matrix-vector multiplications in the LSTM cell and they work under the shadow of the PEs. Their parallelism is configurable to improve the performance and reduce the latency depending on the FPGA resources available. The batch normalization (BN) [103] unit, which is optional and can be turned off via the controller, performs the batch normalization on the results of the matrix-vector multiplications. The Sigmoid/Tanh are the non-linear modules which apply the activation functions. We implement these activation functions using a piece-wise linear approx-

imation [104], which is shown to have little impact on accuracy during LSTM-RNN inference [38]. The outputs will be buffered in the output buffer while waiting to be transferred via DMA.

4.4 Evaluation

4.4.1 Experimental Setup

Many variants of LSTM have been proposed which are suitable for different tasks. In this work, the LRCN [11] for video activity recognition is used to demonstrate our approach. Typically, the LRCN is implemented using a CNN to extract a fixed-length vector of features which are then passed into a recurrent sequence learning module, such as an LSTM. In this work, the features of each frame in the video come from the average pool layer of the Inception-v3 which has been pre-trained on the ImageNet dataset. An additional Fully Connected layer is applied to transfer the features number to 1792 and then fed to our LSTM system. We retrain the LSTM network to get the top-1 accuracy of 72.97% and top-5 accuracy of 89.61% which are higher than the accuracy of 67.37% in the original LRCN design [11].

To recognize the performance and limitations of the proposed LSTM hardware acceleration, we implement the hardware system for the LSTM part in LRCN for the RGB model, where the LSTM-256 model has 256 hidden units. Each LSTM-256 gate weights matrix is 2048*256 and there are four gates. The target platform is Xilinx ZC706, which consists of a XC7Z045 FPGA and dual ARM Cortex-A9 processor. 1 GB DDR3 RAM is installed on the platform as the off-chip memory. The on-chip memory of the XC7Z045 is 19.2Mb while the weights in this LSTM model are more than 32Mb which are too large to store in the on-chip memory of the FPGA. We also implement the LSTM-512 model which has 512 hidden units using the Virtex 7 VX690T FPGA.

Table 4.2: Resource Utilization

		LUT	LUTRAM	FF	BRAM	DSP
Zynq 7045	Avail.	218600	70400	437200	545	900
	Used	165668	49224	150451	517.5	900
	Utili.	75.8%	69.9%	34.4%	94.9%	100%
Virtex7 690T	Avail.	433200	174200	866400	1470	3600
	Used	203549	71478	221576	1070	2060
	Utili.	47%	41%	25.6%	72.8%	57%

4.4.2 Resource Utilization

Table 4.2 shows the resource utilization for our stall-free BPE design on the Zynq 7045 FPGA. The number of PEs, N_{pe}, is configured to 1024 targeting LSTM-256 while the batch size is 64. N_t is 16 when the DMA data bus is 256-bit with a 16-bit LSTM datapath. If the DMA data bus is 512-bit then the proper batch size is 32. N_t needs scaling if DMA data bus works under a different frequency with computation engines. For our system on Zynq, almost all the FPGA's hardware resources are utilized. A few multiplication units are implemented using LUT because there are only 900 DSP elements in our system. Note that the number of PEs, N_{pe}, is configured to 2048 for LSTM-512 targeting Virtex 7 VX690T FPGA because this device has an abundance of DSPs.

The best batch size. The best batch size is determined by balancing the computation time and communication time from the off-chip to on-chip memory. For case 1 and 2, the best batch size on Zynq can be easily calculated from Equation (4.1), which shows that B = $N_{pe}/N_t = 64$. However, for case 3, the performance equations (4.5) and (4.6) are complex, but we can still get 64 as the proper batch size, as illustrated in Fig. 4.5. The performance is not related to B when $B \geq \frac{N_{pe}}{N_t}$ as shown in equations (4.2) and (4.5), which means increasing the batch size does not increase performance beyond a certain point, but only wastes the on-chip memory.

The proper blocking number. For a given LSTM model, when the blocking number increases, the block size decreases, and then the required on-chip memory decreases, because we will only store two blocks on the FPGA. This means that we can process a large LSTM system efficiently even with a small FPGA. However, for a given system, the blocking number cannot

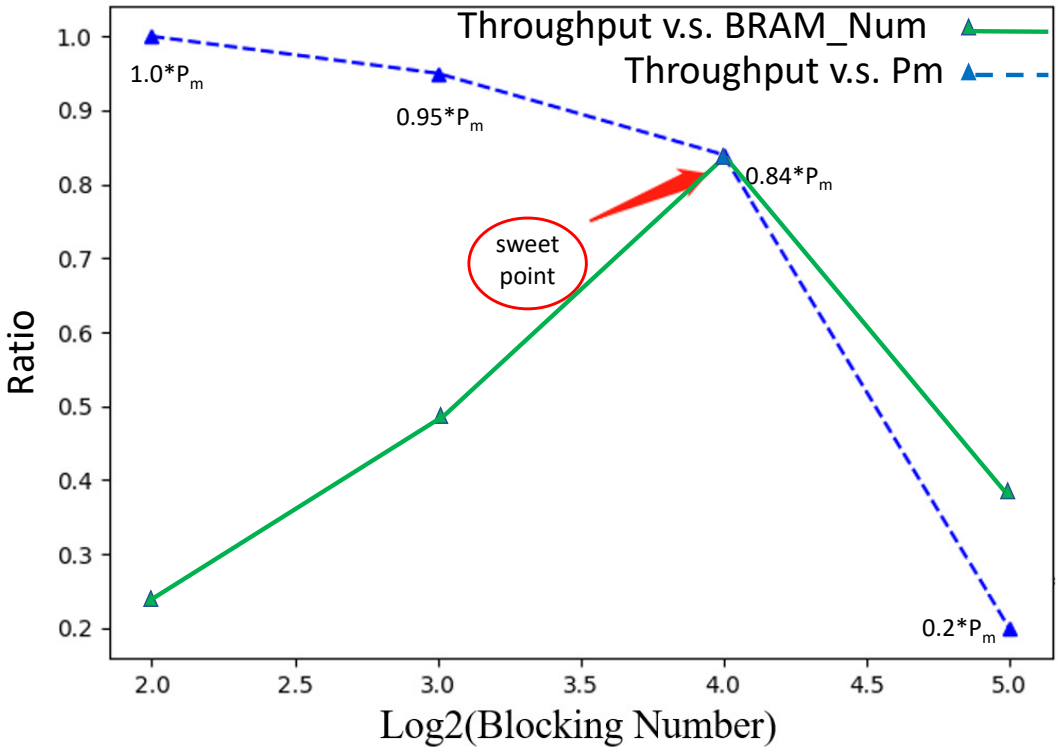


Figure 4.10: Throughput with various blocking numbers on ZYNQ 7045.

be too large because performance can be reduced as shown in case 3. The performance of the LRCN with different blocking numbers on the Xilinx ZC706 platform is shown in Fig. 4.10. P_m is the ideal performance when all the weights are stored in the on-chip memory without external DRAM accesses. It is the highest performance that the system can achieve. From Fig. 4.10, the proper blocking number is 16, which is the sweet point with only 1/8 on-chip memory required compared to previous research which put all the weights in the on-chip memory. It is the best trade-off between on-chip memory size/usage (or FPGA device) and performance. For a given application and performance requirement, the proper blocking number and blocking size will help us to choose the proper FPGA device. We do not need to select a large and expensive FPGA with large on-chip memory before the blocking-batching strategy is applied. When the blocking number decreases from 16 to 8, the performance can still be boosted by about 10%. However, a larger and more expensive FPGA with double on-chip memory will be required. Furthermore, if the user can bear with a reduced performance then they can choose a smaller and cheaper FPGA as shown in Fig. 4.10.

Table 4.3: Performance comparison of the FPGA design versus CPU and GPU.

	CPU	GPU	This Paper	This Paper
Platform	Intel Xeon E5-2665	TITAN X Pascal	Virtex 7 VX690T	Zynq 7Z045
Frequency	2.4 GHz	1.62 GHz	125 Mhz	142 MHz
Technology	22 nm	16 nm	28 nm	28 nm
Power (W)	93	159	26.5	10.6
Precision	32 bit float		16 bit fixed	
Model Size per Frame[1]	8192[1]* 256			
Time per Sample[2] (ms)	14.45	0.78	0.38	0.61
Energy per Sample[2] (mJ)	1343	124.02	10.05	6.47

[1] Combing the four matrices of i, f, o, c gates.

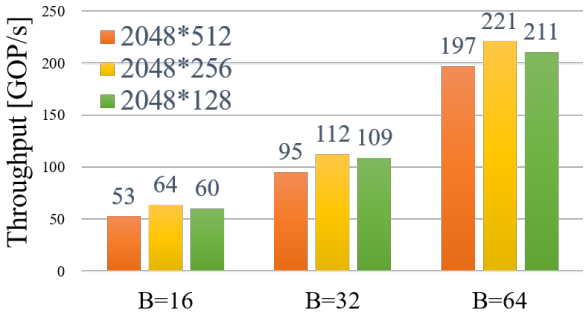
[2] Each sample/video has 32 frames.

4.4.3 Performance and Efficiency Comparison

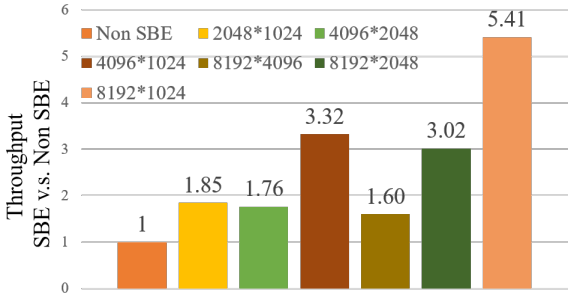
To compare the performance of the proposed design on FPGA with other platforms, we implement the LRCN on Intel Xeon E5-2665 CPU and NVIDIA X Pascal GPU based on Tensorflow(r1.12) framework. The CuDNN 7.4.1 libraries are used for optimizing the GPU solution. Both CPU and GPU implementations run with batch size set to 32 samples, which are 1024 frames in total. Compared with the LRCN on CPU and GPU, our Zynq FPGA design is 23.7 and 1.3 times faster and consumes 208 and 19.2 times less power respectively as shown in Table 4.3.

We have demonstrated parameterizable performance scaling for different LSTM sizes and batch size approaches, see Fig.4.11(left). With very large LSTM models, our design can achieve 1.60-5.41 times higher performance than the ones without SBE, as shown in Fig.4.11(right). In addition, the performance scaling for different blocking number is shown in Fig.4.10. The results show flexible customizability of the architecture for different scenarios.

To illustrate the benefits of our proposed approach, some existing FPGA-based LSTM-RNN accelerator designs are compared with ours in Table 4.4. For a fair comparison, We only show the previous work with detailed implementation of the LSTM system storing the weights in



(a) Throughput with various batch sizes on ZYNQ 7045.



(b) Throughput of our design v.s non SBE design for very large LSTM systems on ZYNQ 7045.

Figure 4.11: Design throughput.

external memory of FPGA. We list the FPGA chips, model storage, precision, run-time frequency, throughput, power efficiency and resource efficiency. The table contains a range of designs across this parameter space for comparison. Our design achieves power efficiency as 20.84 GOPS/W and resource efficiency as 0.246 GOPS/DSP which are the highest with respect to state-of-the-art implementations on FPGAs operating on a dense LSTM model with weights stored in off-chip memory. With a similar number of DSP resources to [19], our system using Virtex 7 achieves 356 GOPS which is the highest performance among all the FPGA implementations of LSTMs storing weights in the off-chip memory. Because of routing congestions, our Virtex 7 design only runs at 125Mhz.

With our weights reusing SBE, small FPGAs can still process a large RNN model efficiently. Note that our comparison does not cover recent approaches [15, 68, 70] about LSTM acceleration using model compression and weight pruning to fit in on-chip memory. Such techniques are orthogonal to our proposed approach. Since useful inference results may not be possible when the FPGA has insufficient memory to store an accurate compressed model, it can still suffer from insufficient on-chip memory of FPGAs for large compressed models. Our technique complements these approaches for improving efficiency. Future work will explore pruning methods to allow large, sparse models to run on FPGAs.

Table 4.4: Comparison with previous implementations of dense LSTM models with weights on off-chip Memory.

	Chang [59]	Guan [38]	ESE [9]	FP-DNN [19]	This Paper	This Paper
FPGA	Zynq 7Z020	Virtex 7 VX485T	Kintex KU060	Stratix V GSMD5	Virtex 7 VX690T	Zynq 7Z045
Technology (nm)	28	28	20	28	28	28
Model Storage	off-chip					
Prec. (bits)	16	32[a]	12	16 32[a]	16	16
DSP Number	220	2800	2760	3180[c]	3600	900
Freq. (Mhz)	142	150	200	150	125	142
Perf. (GOPS)	0.47	7.26	282[b]	316 86[a]	356	221
Power Effi. (GOPS/W)	0.268	0.37	6.87	12.63 3.44[a]	13.48	20.84
Resource Effi.[d] (GOPS/DSP)	0.002	0.003	0.102	0.099 0.027[a]	0.099	0.246

[a] Floating point
[b] Dense Model
[c] One Intel FPGA DSP includes two 18*18 multipliers
[d] To make a fair comparison, the total number of DSP in device is used to calculate GOPS/DSP when evaluating LSTM accelerator

4.5 Summary

This chapter presents a blocking-batching strategy to optimize the throughput of large RNNs that are too large to fit in on-chip memory on FPGAs. This is achieved by reusing the RNN weights and eliminating data dependencies and stalls through a stall-free hardware architecture. A performance analysis based on LSTM models is also presented to balance area, power, and performance in FPGA designs. When compared to the state-of-the-art design [19] that use off-chip memory to store weights, our design on Zynq achieves 1.65 times higher performance-per-watt efficiency and 1.60 times higher performance-per-DSP efficiency. When compared to CPU and GPU implementations, our hardware architecture is 23.7 and 1.3 times faster while consuming 208 and 19.2 times less energy, respectively.

Chapter 5

Low Latency RNNs on FPGAs

5.1 Introduction

This chapter presents a novel reconfigurable partially-folded architecture for reducing the latency of RNNs, with gravitation wave detection as an example application. Among the many RNN variants, the most popular one is LSTM. We also propose to balance initiation intervals in a multi-layer LSTM network, by identifying appropriate reuse factors for each layer, to improve hardware efficiency. Initiation intervals represent the time intervals between the start times of different stages in a pipeline. The performance of the pipeline is dictated by its slowest stage. Meanwhile, the reuse factor corresponds to the number of times a multiplier is used in the computation of a module. Further details will be elaborated on in the following sections.

The detectors at the Laser Interferometer Gravitational-Wave Observatory (LIGO) produce time-series data, as they capture cosmic events such as black hole mergers which happen at unknown times and of varying durations. Accelerating RNN inference using reconfigurable accelerators such as FPGAs would enable sophisticated processing, such as anomaly detection, to run in real time on the data stream from the detector and generate a fast response.

However, existing LSTM accelerators cannot support low-latency and effective multi-layer execution, especially when targeting small LSTM models with requirements of low latency and

high throughput for scientific applications. Many existing FPGA-based LSTM accelerators are designed with the same idea as their GPU counterparts, which utilize a single computational engine architecture where the engine is designed to run one block or layer at one time, and the whole network is processed by running the engine repeatedly [1, 2]. Their design consists of arranging computing resources to form a single core with many processing elements, leveraging data level parallelism. However, when the size of the targeted LSTM layer is small, these hardware resources will not be fully utilized, e.g., when targeting a small LSTM layer, the Brainwave hardware utilization is lower than 1% [1], while the utilization of the NPU can be lower than 15% [2]. Moreover, since a single engine is used, the various layers must have the same amount of parallelism which is not flexible to take full advantage of the customizability of FPGAs. Thus, this work applies a partially-folded architecture to map all the LSTM layers on-chip and perform the computation for different layers on their own unit with independent optimization to achieve low latency and high system throughput.

Unlike CNN inference designs [105, 106] which only have forward datapaths and can be fully pipelined, there are feedback datapaths in RNN inference and data dependencies exist between the current timestep and the next timestep. Unrolling the timesteps fully may help, however the sequence length (timestep) of an LSTM model is usually larger than the number of layers [25], e.g., 1500 timesteps in an LSTM layer in DeepSpeech [27], which makes the full unrolling of timesteps impractical on FPGAs because of the limited hardware resources.

To accelerate an RNN model with multiple LSTM layers, this work proposes coarse grained pipelining with balanced II (initiation interval) to improve system throughput and reduce latency. This is achieved by identifying appropriate reuse factors for each layer, resulting in fast response and enhanced resolution for processing sensor data. It can achieve the best (smallest) system level II for a neural network with multiple LSTM layers on a given FPGA. The II is the number of clock cycles before a unit can accept new inputs and is generally the most critical performance metric in systems [107]. A perfect pipeline has $II = 1$ cycle, as this is required to keep all pipeline stages busy. However, the II of an LSTM layer is generally larger than one because of the data dependencies. For a model with multiple layers in sequence, the initiation interval of this model is decided by the largest II among all the layers [108], as shown in Fig. 5.1.

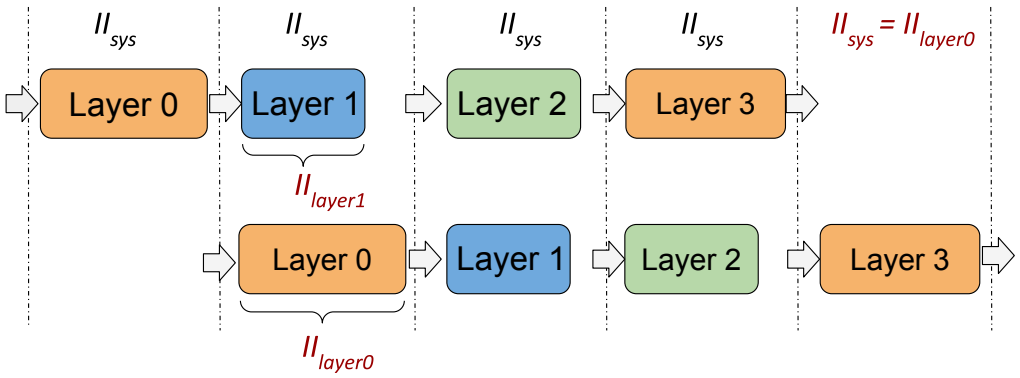


Figure 5.1: Unbalanced layer IIs among various cascaded layers in an RNN model.

The unbalanced IIs in various layers result in hardware inefficiency and low throughput. Accelerating a deep LSTM model is challenging since the computation load varies greatly among layers and data dependency exists both time-wise and layer-wise.

Our approach is to ensure all the layer IIs are balanced to eliminate system stall, so that the system becomes a coarse grained seamless pipeline. It increases pipeline parallelism by performing more computations without increasing latency, and without introducing additional memory traffic or storage. Unbalanced IIs in a pipeline is a common issue, but few studies address balancing IIs in the context of accelerating multi-layer DNNs, especially for RNNs/LSTMs. The proposed coarse-grained pipelining is similar to layer parallelism but the granularity in our approach does not need to cover an entire layer. An LSTM layer can still be divided into multiple blocks with pipeline parallelism. In addition, a customizable template for this architecture has been designed, which enables the generation of low-latency FPGA designs with efficient resource utilization using high-level synthesis (HLS) tools. Moreover, We develop an optimization algorithm such that, given the dimensions of the LSTM layers and a resource budget, computes a partitioning of the FPGA resources for an efficient low-latency design.

We make the following contributions:

- A novel technique for balancing IIs of multi-layer LSTM inference to increase hardware efficiency and design throughput.
- A low latency LSTM template which enables the generation of low-latency FPGA designs

Table 5.1: System Parameters

II_{sys}	System initiation interval
TS	Timestep number
ii_N	Timestep loop initiation interval in the LSTM layer N
II_N	Initiation interval for layer N
LT_N	Latency of a single timestep loop for layer N
LT_α	Latency of the unit α; α could be mult / mvm / tail / σ
x_t	The input vector x at timestep t
h_t	The hidden vector h at timestep t
Wx	LSTM gates weight matrix for input vector.
Wh	LSTM gates weight matrix for hidden vector.
Lx	Number of elements in the input vector x
Lh	Number of elements in the hidden vector h
R_x	Reuse factor for MVM involving LSTM input vector x_t
R_h	Reuse factor for MVM involving LSTM hidden vector h_t
R_t	Reuse factor for LSTM tail unit

with efficient resource utilization by HLS tools. We open source the templates with some examples[1].

The specific RNN layered structure and coefficients are LIGO specific, but the need for low latency would benefit many other applications, especially those requiring real-time response, e.g., low latency would benefit the Large Hadron Collider (LHC) physics [109, 110], adaptive radiotherapy [111] and electronic trading [112]. The proposed techniques can be adapted to address these other applications. Besides, the balancing II technique has been applied to low latency graph neural network designs [34, 35, 113].

5.2 Design and Optimization

This section analyzes unbalanced II issues and introduces several optimizations for multi-layer RNN designs. We define a few parameters, as shown in Table 5.1 for later calculations.

[1]https://github.com/walkieq/RNN_HLS

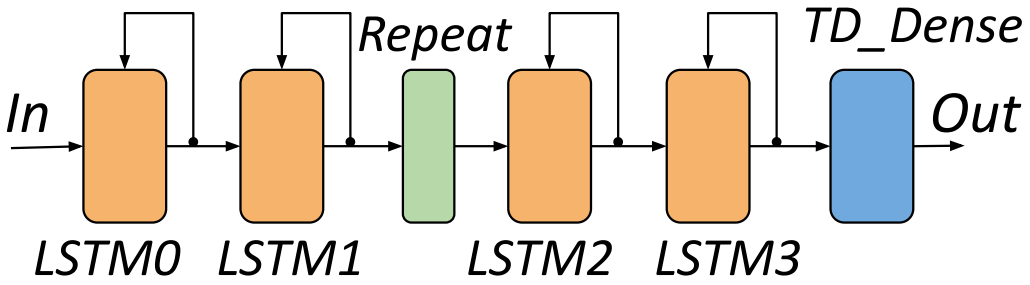


Figure 5.2: Overview of the LSTM-based autoencoder.

5.2.1 LSTM-based Autoencoder for Gravitational Wave Detection

Fig. 5.2 shows an overview of the LSTM-based autoencoder used for gravitational wave detection. The models and the dataset are available on GitHub [114, 26]. The autoencoder consists of two components, an encoder and decoder. The encoder learns to transform data from the input layer into a latent-space representation, which acts as a data "bottleneck". The decoder then reconstructs the output of the reduced latent representation as close as possible to its original input. When the error between input and reconstructed values is high, the input is flagged as anomalous. In this work, an LSTM-based autoencoder is used as an unsupervised prediction model to detect the anomalies for gravitational waves. This works by only training the LSTM-autoencoder to encode and decode normal background conditions at the LIGO interferometers. When an event containing a gravitational wave passes through the autoencoder, the model cannot encode and decode the additional strain provided by the gravitational wave. Both the encoder and decoder have two LSTM layers. A TimeDistributed dense layer is applied before the data output.

5.2.2 System II for Multi-layer LSTM Networks

Accelerating a deep LSTM model which has multiple layers is challenging since the computation varies greatly among layers and data dependencies exist both time-wise and layer-wise. An efficient technique to improve throughput and reuse computational resources is to pipeline hardware units. If each input can overlap with itself, we can achieve simultaneously inference

parallelism within a run by coarse grained pipelining as shown in Fig. 5.1.

However, a naive implementation can result in a large number of idle cycles due to inter-layer dependencies since the pipeline is not seamless; a particular layer might stall until the previous layer finishes. The unbalanced IIs in various layers results in hardware inefficiency and low system throughput. Typically, the particular layer with the largest II should be optimized since it dominates the system II. Generally, the II cycles can be reduced if more hardware resources are allocated to that particular layer by adding more parallelisms. So the targeted layer should be allocated as many hardware resources as possible. However, the hardware resources on a given FPGA is limited, which means that the other layers may occupy less hardware resources. When the resources for a layer decrease, the II of that layer will increase. Then this layer may become the one that has the largest II and dominates the design. Thus, the optimal case is that all the layers have the same II, in which scenario the design utilizes the hardware resources efficiently and achieves the highest system throughput as shown in Fig. 5.3.

Besides, we find that we do not need to unroll every unit in order to achieve the lowest II. Some hardware resources can be saved from the units which do not require full unrolling. Then these saved hardware resources can be reallocated to the other units which dominate the system to achieve low initiation intervals. As shown in Fig. 5.3, the hardware resources for layer 1 can be reduced so that the saved resources can be reallocated for layer 0. The II_{layer1} is increased to II'_{layer1} while the II_{layer0} which is the largest can be reduced to II'_{layer0} so that the final system II_{sys} can be reduced.

Partitioning FPGA resources to enhance throughput has been studied for CNNs [105, 106, 115, 116] but they do not touch the RNNs and the recurrent nature as well as the data dependencies in RNN computations, which are absent from CNNs. We develop an optimization algorithm such that, given the dimensions of the LSTM layers and a resource budget, computes a partitioning of the FPGA resources for an efficient and balanced high-performance design. Our algorithm runs in seconds and produces a set of reuse factors [109]. We then use these factors to parameterize an LSTM template design specified using HLS to form a complete multi-layer LSTM implementation. Since all the layers have the same II, we only need to focus on the

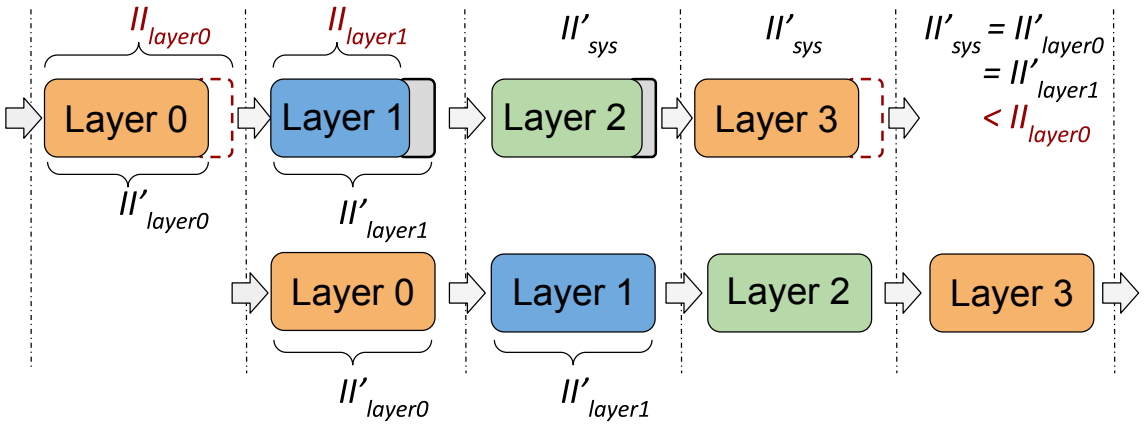


Figure 5.3: Overview of the method used to balance IIs.

optimization for a single LSTM layer. The layer II and system II are

$$II_N = ii_N \times TS \tag{5.1}$$

$$II_{sys} = \max(II_0,\ II_1,\ ...,\ II_N) \tag{5.2}$$

The original II_N should be $II_N = ii_N \times TS + (LT_N - ii_N)$. However, the extra $(LT_N - ii_N)$ cycles can be eliminated after using the rewind #pragma in for Vivado HLS. The rewind pragma enables rewinding, or continuous loop pipelining with no pause between the end of one loop iteration and the start of the next iteration. So the proposed balancing method has two benefits. First, it improves throughput due to pipelining. Second, it reduces system latency since if the LSTM loop initiation interval, ii_N, can be reduced by 1 cycle, then the system latency can be reduced by TS cycles in total according to Equation (5.1).

5.2.3 The II of a Single LSTM Layer

Algorithm 5.1 illustrates the pseudocode of an LSTM layer. Wx and Wh denote the LSTM weights for input and hidden vectors. B represents the bias. x is a set of input vectors and has the size of $(TS,\ Lx)$. h is a set of hidden vectors and has the size of $(TS,\ Lh)$. Seq decides if the whole sequence of hidden vectors should be returned or just the one in the last timestep.

The function $MVM\_x()$ performs MVM operations and the addition of bias for the LSTM

Algorithm 5.1: The pseudocode of an LSTM layer.

```
1  Function LSTM_layer(Wx, Wh, B, x, Seq):
2      h_0 ← 0;
3      for t = 1 to TS do
4          Acc = MVM_x (Wx, B, x_t);
5          Acc = MVM_h (Wh, Acc, h_{t-1});
6          Acc = Sigmoid_tanh (Acc);
7          h_t = LSTM_tail (Acc);
8      if Seq then
9          return h;                          ▷ A set of all the hidden vectors
10     else
11         return h_t;                        ▷ The hidden vector at the final timestep
12 End Function
```

gates involving the input vectors. The MVMs involving the hidden vectors are conducted in the function $MVM\_h()$. The $Sigmoid\_tanh()$ is the activation function which performs sigmoid or hyperbolic tangent operations. The $LSTM\_tail()$ function contains the element-wise operations as shown in Fig. 2.2. The result h_t as labeled in red is required in $MVM\_h()$ in the next timestep iteration, which shows the existence of data dependencies.

This work splits one LSTM layer into two sub-layers. The first one is the mvm_x which has no data dependencies and performs MVM operations for the LSTM gates involving the input vectors while the second one includes all the others which form a loop with data dependencies, as shown in Fig. 5.4. For accelerating LSTM layers used for gravitational wave detection, the system is designed to achieve the average latency (system II) as small as possible. To achieve the lowest system II, fully unrolling the neural network model is an effective method which utilizes a multiplier only once in the computation of a layer. E.g., a fully connected (FC) layer with input size num_in and output size num_out can achieve the lowest latency if there are $num\_in \times num\_out$ multipliers. This is the most parallel and fast way a layer can be computed. It has been demonstrated in the HLS4ML based DNN designs for particle physics [109]. However, unlike forward computation in the FC layers used in the design of [109], there are data dependencies in LSTM computations.

After we have split the LSTM layer into two sub-layers, the two can be pipelined as shown in Fig. 5.5. According to the discussion in Section 5.2.2, the optimal case is when the two

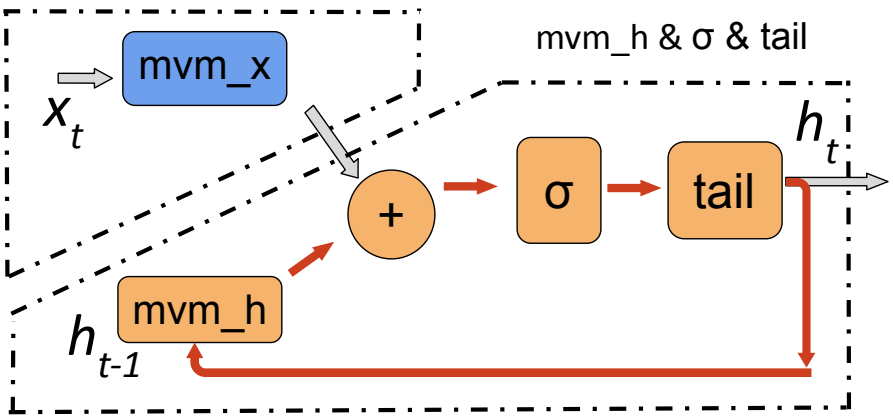


Figure 5.4: An LSTM layer after performing the transformation.The details of LSTMs can be found in the background Section 2.1.2 and Fig. 2.2.

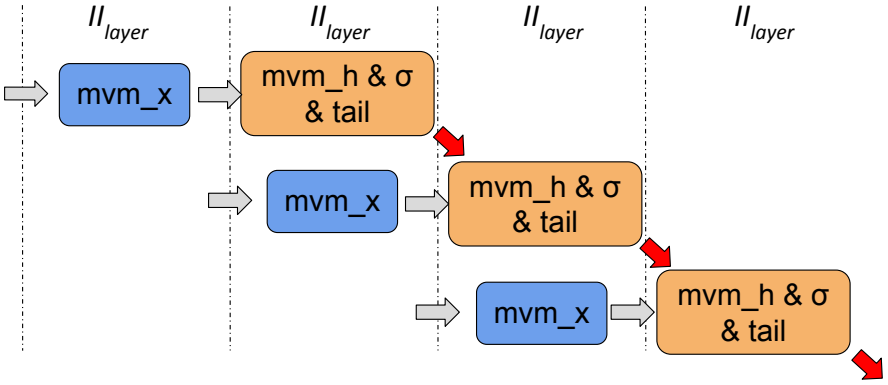


Figure 5.5: Coarse grained pipelining in an LSTM layer.

sub-layers have the same II. Since the second sub-layer is complex and its II is usually larger than the one of the first sub-layer, the parallelism for the first sub-layer does not need to be as large as possible, resulting in a reduction of the number of multipliers needed to process the *mvm_x* unit. The saved multipliers can then be reallocated for other layers to achieve a lower system II. Reducing the parallelism of *mvm_x* does not hurt the system latency. Normally, each input vector can finish the calculation in the shadow region of processing the h_t because of the pipelining. Besides, the cycles for processing the first *mvm_x* can be eliminated when calculating the layer II because of the keyword of rewind in Vivado HLS.

While the second sub-layer may seem complex, if the design is split into more sub-layers, these sub-layers cannot be coarse grained pipelined. The reason is that the start of the next iteration needs the result from the current iteration, as shown by the red arrows in Fig. 5.5

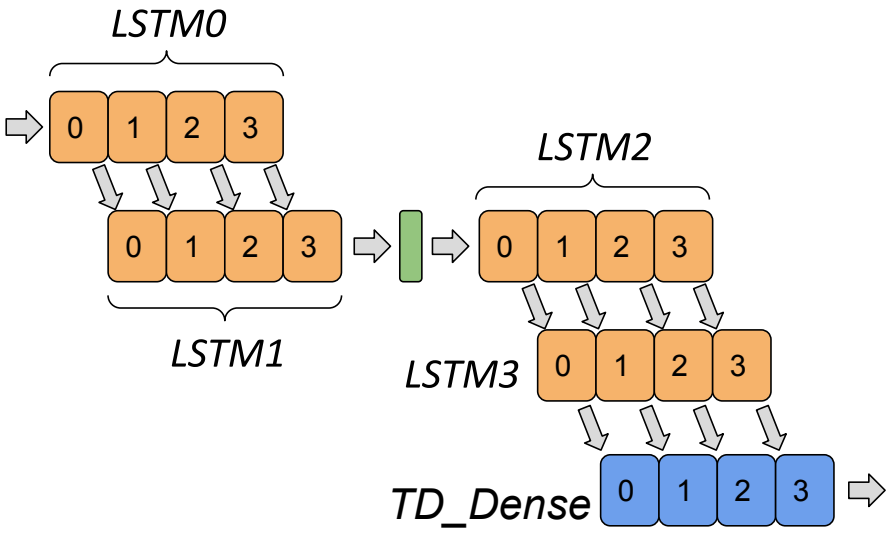


Figure 5.6: Timestep overlapping.

5.2.4 Overlapping the Computations in Cascaded LSTM Layers

In the proposed coarse grained pipelining, the processing of the cascaded LSTM layers can be overlapped. The second layer does not need to wait for the whole sequence of hidden vectors to be ready. Just one hidden vector from the former LSTM layer is sufficient to start the calculation of the next LSTM layer as shown in Fig. 5.6. It helps to reduce the overall system latency. It has to be noted that the LSTM2 can only start after the LSTM1 calculation is completed, since only the last timestep hidden vector is returned in LSTM1, which is decided by the structure of the autoencoder.

5.3 Implementation

5.3.1 HLS Implementation

This work maps all the layers on-chip and different layers run in a fashion of coarse grained pipelining to increase the system throughput. Besides, this work always seeks to achieve extremely low latency by utilizing as many hardware resources as possible. However, because of the data dependencies between different timesteps in LSTM calculation, the initiation in-

terval is typically larger than 1. In this case, HLS will automatically increase the initiation interval until it can find a feasible schedule. For complex designs it is common to partition functionality into multiple modules, streaming data between them through explicit interfaces. Smaller components are more modular, making them easier to reuse, debug and verify. The effort required by the HLS tool to schedule code sections increases dramatically with a large number of operations that need to be considered for the dependency and pipelining analysis. Scheduling logic in smaller chunks is thus beneficial for compilation time and sometimes also for system latency. Our experiments show that inlining every function, especially the mvm_x and mvm_h in the LSTM gates, brings large II when the involved matrices are large.

The trade-off between latency, throughput and FPGA resource usage is determined by the parallelization of the inference calculation. This work adopts the reuse factor used in [109] to fine tune the parallelism, which is configured to set the number of times a multiplier is used in the computation of a module. In one extreme, all multiplications can be performed simultaneously using a maximal number of multipliers, while alternatively in the other extreme, one can use only one multiplier and perform the multiplications sequentially; between these extremes the user can fine tune algorithm throughput versus resource usage.With a reuse factor of one, the computation is fully parallel. With a reuse factor of R, $\frac{1}{R}$ of the computation is done at a time with a factor of $\frac{1}{R}$ fewer multipliers.

The total number of multiplications required to infer a given LSTM layer using 16-bit is:

$$DSP_{layer} = \frac{4 \times Lx \times Lh}{R_x} + \frac{4 \times Lh^2}{R_h} + 4 \times Lh \tag{5.3}$$

$$DSP_{model} = \sum_{layer=1}^{N} DSP_{layer} \leq DSP_{total} \tag{5.4}$$

Compared with the number of multipliers used in LSTM gates, the one required in the LSTM tail unit is small so the R_t is set to 1. Otherwise, $\frac{4 \times Lh}{R_t}$ should be used in Equation (5.3). Besides, since the LSTM cell status, c_{t-1}, is represented in 32-bit, the $f_t \times c_{t-1}$ in the LSTM tail needs two Xilinx DSPs to implement one multiplier. Thus, the LSTM tail unit consumes $4 \times Lh$ DSPs. The activation function sigmoid is implemented using BRAM-based lookup tables

with a range of precomputed input values. The hyperbolic tangent function is implemented as piecewise linear function [117] to reduce the latency. In the next subsection, we introduce our method for determining R_x and R_h with a given FPGAs.

5.3.2 Design Space Exploration

FPGA multipliers are pipelined; therefore, the latency of one MVM computation, LT_{mvm}, is approximately

$$LT_{mvm} = LT_{mult} + (R - 1) \times II_{mult} \tag{5.5}$$

where LT_{mult} is the latency of the multiplier, II_{mult} is the initiation interval of the multiplier, which is one cycle in this work. Equation (5.5) is approximate because, in some cases, additional cycles could be introduced for signal routing. Besides, the Vivado HLS tool will replace a multiplier by an adder when the corresponding weight is simple.

As we discussed in Section 5.2, the optimal case is that the two sub-layers in an LSTM layer have the same II, which results in Equation (5.6).

$$II_{sublayer} = LT_{mvm\_x} = LT_{mvm\_h} + LT_{\sigma} + LT_{tail} \tag{5.6}$$

where LT_{mvm_x} and LT_{mvm_h} are the latencies of the MVM units involving input vectors x and hidden vectors h respectively. LT_{σ} is the latency of the sigmoid function and LT_{tail} is the latency of the LSTM tail unit. These units are shown in Fig. 5.4. If we substitute Equation (5.5) into Equation (5.6) and then we get

$$R_x = R_h + LT_{\sigma} + LT_{tail}. \tag{5.7}$$

The architecture designed in this section serves as a baseline to deploy our methodology, whose goal is to find Pareto-optimal sets of reuse factors of the proposed accelerator to achieve a good trade-off between our design objectives, which are hardware resources, energy, and per-

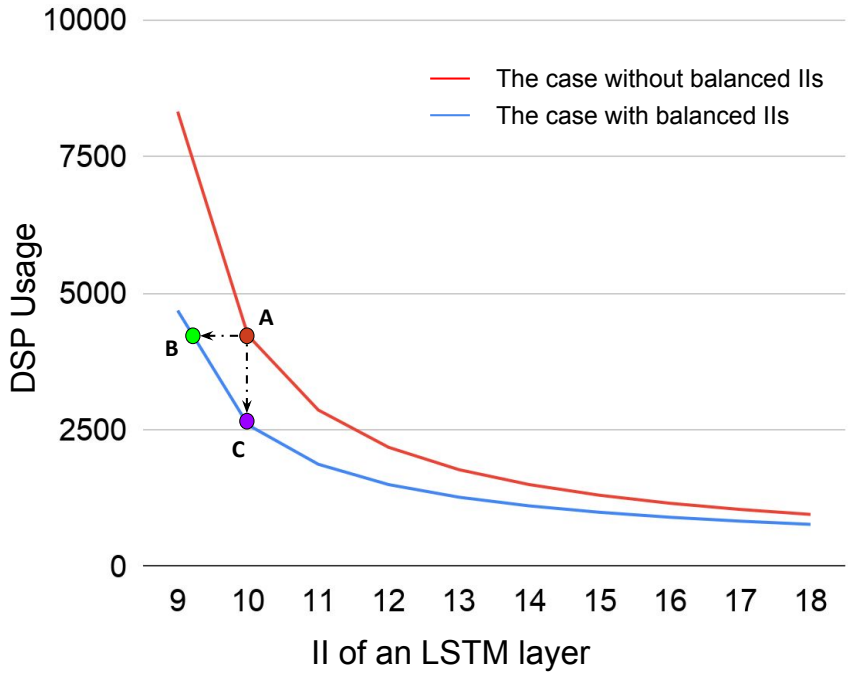


Figure 5.7: Pareto frontier.

formance. To achieve low latency, the reuse factors should be as small as possible since when they decrease the parallelism increases, leading to high throughput. However, when reuse factors decrease, the required hardware resources increase and may easily exceed the number of total hardware resources on an FPGA. If we substitute the Equation (5.7) and Equation (5.3) into Equation (5.4), we can get a quadratic inequality of R_h, which gives the minimum R_h for a given number of DSPs.

Fig. 5.7 illustrates the exploration results of an LSTM layer with $(Lx, Lh) = (32, 32)$ and different values of reuse factors, which are from 1 to 10. The red line represents the cases with the same R_x and R_h. The blue line shows the cases with balanced IIs, where R_x and R_h meet the constraint in Equation (5.7). For simplicity, LT_σ is set to 3 and the LT_{tail} is 5. Please note that LT_σ and LT_{tail} are both system dependent and can vary depending on clock frequency and FPGA devices. After balancing IIs, the Pareto frontier moves from red line to blue line. With the proposed technique, we can achieve a same II with less DSP usage (from point A to point C) or we can achieve a better II (from point A to point B) as shown in Fig. 5.7.

5.4 Evaluation and Analysis

This section presents the performance of the RNN models developed for gravitational wave detection on two generations of Xilinx FPGAs (ZYNQ 7045 and U250) demonstrating the scalability of the proposed optimization. Details can be found in Section 5.4.3.

5.4.1 Experimental Setup

Simulated gravitational waves (GWs) are generated using the GGWD library [118]. Noise is generated at a specified power spectral density (PSD) to mimic normal detector background conditions using PyCBC [119]. This approach to simulated data generation ignores glitches, blips, and other transient sources of detector noise, though this algorithm can be re-purposed for identifying these detector glitches with unsupervised methods. Signal events are generated simulating GW production from compact binary coalescences using PyCBC [119], which itself uses algorithms from LIGO's LAL Suite [120]. Signal events containing GWs were created overlaying simulated GWs, with the SEOBNRv4 Approximant, on top of detector noise. This provides an analogous situation to a real GW, in which the strain from the incoming wave is recorded in combination with the normal detector noise. Data are then whitened and band-passed, then normalized. The training set has 240K gravitational wave events. The validation set and test set have 60k and 50k events respectively. To study the performance and limitations of the proposed optimizations and hardware architecture, the designs are implemented using Vivado HLS 19.2. Two generations of Xilinx FPGAs, the ZYNQ 7045 and U250, are evaluated and compared with previous work.

5.4.2 Model Accuracy

To quantify the performance of the autoencoders for anomaly detection implemented by various neural networks, we use the AUC metric, or area under the Receiver Operating Characteristic (ROC) curve, as shown in Fig. 5.8, with higher AUC corresponding to better performance. The

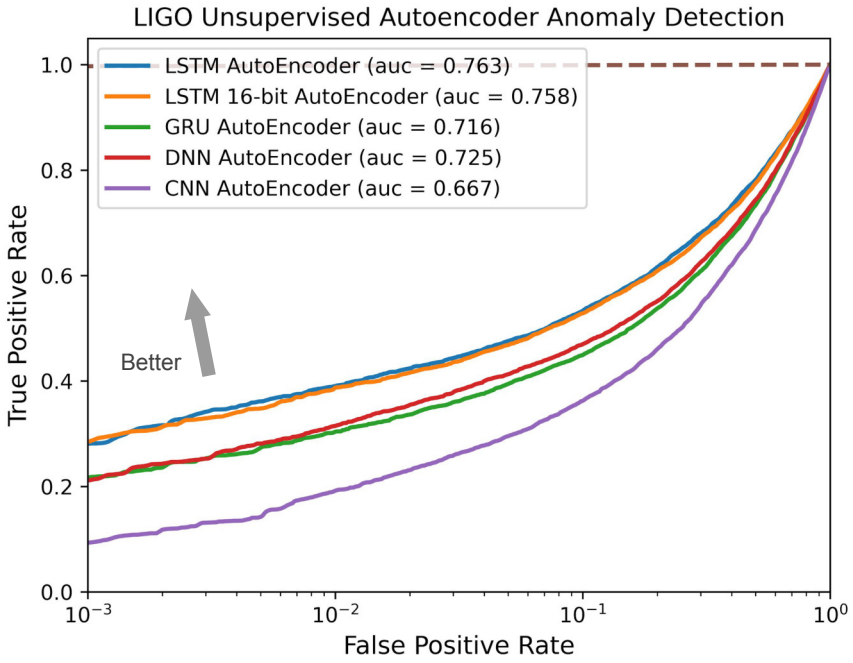


Figure 5.8: AUCs and ROC curves for various autoencoders.

default timestep [114] of 100 is used. AUC is a common metric for evaluating models as it is classification-threshold-invariant. The threshold for flagging an anomaly by its loss spike can be calculated by setting a false positive rate (FPR) on noise events. The higher the threshold for detecting an anomaly, the lower the FPR will be. This threshold can be used to calculate the corresponding true positive rate (TPR) on signal events. We observe that the LSTM-based autoencoder has the highest AUC, and hence the best performance, among the unsupervised designs [114] with various NN layers, including GRU, CNN and DNN. Additionally, Qkeras [121] is used to quantize the LSTM-based autoencoder to 16-bit. We find this precision to have a negligible effect on the NN performance.

5.4.3 Performance and Efficiency Comparison

To illustrate the benefits of our proposed approach, two LSTM-based autoencoders are evaluated. The first one is a small autoencoder which has the same architecture as the one used in gravitational wave detection described in Section 5.2.1 but only has two LSTM layers, each having 9 hidden units. The results are shown in Table 5.2. It is running at 100MHz with 8

Table 5.2: Performance comparison of the FPGA designs.

	Z1	Z2	Z3	U1	U2	U3
FPGA	Zynq 7045			U250		
DSP total	900			12,288		
R_h	1	2	1	1	1	4
R_x	1	2	9	1	9	12
LUT used	45k (21%)	45k (21%)	43k (20%)	449k (26%)	463k (27%)	516k (30%)
DSP used	1,058 (118%)	578 (64%)	744 (83%)	11,123 (91%)	9,021 (73%)	2,713 (22%)
ii_{layer} cycles	9	10	9	12	12	13
II_{layer} cycles	72	80	72	96	96	104

timesteps. The weights and input are 16 bits. The bias and LSTM cell status are both 32 bits to keep the accuracy. To achieve the lowest latency, the reuse factors should be set to one so that all the operations are unrolled, e.g., the design Z1 in Table 5.2. However the required number of DSPs exceed the one of the total DSPs on this FPGA. One may increase the re-use factor from one to two to fit the design into this FPGA device. However the cost is that now the timestep loop initiation interval, ii_{layer}, increases by one cycle which results in TS cycles increase for the layer II, e.g., the design Z2 in Table 5.2. However, it is not necessary to fully unroll all units in order to achieve the lowest latency. Some hardware resources can be saved from the units which do not require full unrolling and can be allocated to the other units which are dominating to achieve low latency.

With the proposed balancing of IIs, some of the DSPs resources can be rearranged from implementing mvm_x to mvm_h to achieve lower latency, e.g., the design Z3. So this design can still achieve the lowest II like the case with full unrolling, and it is still able to fit in this FPGA device as shown in Table 5.2, showing the benefits of balanced IIs. Besides, with heterogeneous reuse factors, the parallelism of the design can be fine-tuned to make the trade-off between latency, throughput and FPGA hardware resources as shown in Fig. 5.9. With the balanced II, the number of DSPs can be reduced up to 42% while achieving the same IIs.

Besides, to show the adaptability of our technique, the nominal autoencoder [114] developed for

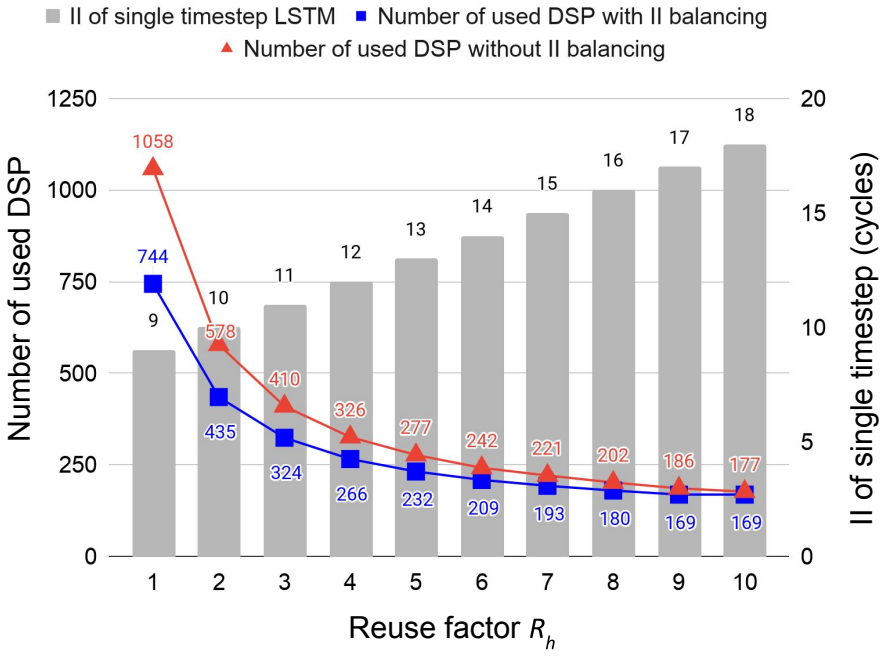


Figure 5.9: Initiation intervals and DSP numbers using various reuse factor R_h on Zynq 7045.

gravitational wave detection is implemented using a larger FPGA, U250, running at 300MHz with 8 timesteps. It has four LSTM layers which have a number of hidden units equal to 32, 8, 8, 32 respectively and one TimeDistributed dense layer before the output. Since the U250 has 12,288 DSPs, the whole fully unrolled autoencoder can be fit into this FPGA with both R_x and R_h set to one, shown as the design U1 in Table 5.2. With our technique of balancing IIs, the DSPs of the design U2 can be reduced by 2102 while achieving the same design IIs and same design throughput. After HLS synthesis, the II is slightly larger than the one estimated by the performance model since the DSP usage is very high and some additional cycles are incurred for signal routing. The design U3 is an interesting version with reuse factors (R_h, R_x) as (4, 12). It achieves a slightly worse II, as shown in Table 5.2, however it consumes 3.3 and 4.1 times less DSPs than design U2 and design U1 respectively. Sometimes, the user may only care about the latency of the LSTM running on the FPGAs, then they can just take the point that gives them the lowest latency with most resources. However, if the user can bear with a slightly reduced latency then they can choose a smaller and cheaper FPGA as shown in Table 5.2. One can choose between using less resources but increasing latency and vice versa. Please note because of the data dependence, the ii_{layer} could be hard to optimize to 1. However, it could be further optimized to a smaller value using fast multipliers or fast activation functions. We leave that

Table 5.3: Latency comparison of the FPGA design versus CPU and GPU.

	CPU	GPU	This work
Platform	Intel E2620	TITAN X	U250
Process	32nm	28nm	16nm
Precision	F32	F32	16 Fixed
Latency	39.7 ms	32.1 ms	0.867 us

for future work since it has a limited impact on the conclusions we draw from our study in this work.

To compare the performance of the proposed design on FPGA with other platforms, we implement the same LSTM-based autoencoder on Intel CPU and NVIDIA GPU. The AVX2 vector instructions are enabled for the CPU while the CuDNN 7.4.1 libraries are enabled for the GPU. Compared with the designs running on CPU and GPU, our FPGA design runs much faster, as shown in Table 5.3. We are processing each inference sequentially (batch 1) since requests need to be processed as soon as they arrive. The GPUs provide large throughput by running many parallel inferences but may not perform well when the batch is small, especially there are data dependencies in LSTMs. However, FPGAs work fast on a single inference with a fully unrolled tailor-made design.

Some other HLS-based RNN/LSTM accelerators for low latency designs on FPGAs are compared with ours in Table 5.4. Rao *et al.* [122] implement an HLS-based low latency LSTM network on an FPGA, tailored specifically for a particle physics application known as Top Tagging. They are the first to integrate the LSTM network into the HLS4ML framework [109]. Lee *et al.* [123] introduce a latency-optimized LSTM design for accurate radio frequency spectral prediction in real-time. Besides, they present a flexible LSTM module generator which can generate optimized LSTM inference cores of arbitrary size and arbitrary fixed-point precision. In this table, we focus on latency since the throughput, power or power efficiency of the other designs are not reported. Our design achieves 4.92 to 12.4 times lower latency compared to the state-of-the-art FPGA designs targeting anomaly detection. Our single-layer design, with a similar amount of DSP resources to another design [122], is 3.9 times faster as shown

Table 5.4: Comparison with previous FPGA-based LSTM designs for anomaly detection and physics

	[123], 2018	[122], 2020	This work	This work
FPGA	Kintex7 K410T	KU115	U250	U250
Process (nm)	20	20	16	16
Model	Single Layer	Single Layer	Single Layers	Four Layers
Application Domain	Anomaly Detection	Physics	-	Anomaly Detection
LSTM hidden units Lh	32	16	32	32,8,8,32
DSPs	1091	2374	2221	9021
Precision (bits)	16 fixed	16 fixed	16 fixed	16 fixed
Frequency (MHz)	155	200	300	300
Latency (us)	4.27	1.35	0.343	0.867

in Table 5.4. Since the FPGAs adopt different transistor technology, performance of designs in [122, 123] should be given a factor of 20/16=1.25 improvement. So the latency of the 2 designs become $3.42\mu s$ and $1.08\mu s$, still worse than the designs from this work. Note that because of the structure of an autoencoder, the processing of the encoder and the decoder cannot be overlapped, which increases the end-to-end latency of the design. Nevertheless, we still achieve better latency than the others which contain only one LSTM layer. Moreover, while the other designs report Vivado HLS synthesis latency, we report the RTL co-simulation latency which is likely to be more accurate.

5.5 Summary

This chapter introduces a partially-folded architecture for RNNs that maps all the layers on-chip and performs computation on their own units with dedicated optimization to achieve low latency and high system throughput. It targets small-sized RNN models with requirements for low latency and high throughput, such as for scientific applications. A novel technique for balancing IIs in multi-layer RNN inference is also presented, which improves hardware efficiency and increases design throughput. In addition, a low latency LSTM template is proposed, which

enables the generation of low-latency FPGA designs with efficient resource utilization by HLS tools. Results show latency reduction of up to 12.4 times over the existing FPGA-based LSTM design.

Chapter 6

Conclusion

6.1 Summary of Achievements

This thesis focuses on optimizing reconfigurable accelerators for RNNs on FPGAs. The objective is to improve the performance and efficiency of FPGA-based RNN designs. This thesis presents three main contributions: a latency-hiding architecture that utilizes column-wise matrix-vector multiplication with a flexible checkerboard tiling strategy, a blocking-batching strategy to reuse RNN weights to optimize the throughput of large RNNs that cannot fit into on-chip memory, and a low latency design of RNNs on FPGAs based on a partially-folded architecture.

Chapter 3 (Contribution 1) presents a column-wise MVM approach for RNNs to eliminate data dependencies and introduces a latency-hiding hardware architecture with hybrid kernels and Configurable Adder-tree Tail (CAT) units, increasing the hardware utilization and system throughput. It also introduces a flexible checkerboard tiling strategy that supports EP and VP to exploit the available parallelism while increasing hardware utilization. In addition, the (EP, VP) parameter space is comprehensively explored. The proposed approach and optimizations are applied to RNN workloads from the DeepBench suite [27]. Compared to the Brainwave design [1] and the Brainwave-like NPU [2] with the same RNN workloads on FPGAs, our design achieves 3.7 to 14.8 times better performance and has the highest hardware utilization.

The evaluation results show a significant speed-up over CPU, GPU and FPGA implementations in related work.

Chapter 4 (Contribution 2) introduces a blocking-batching strategy that reuses the RNN weights to optimize the throughput of large RNNs that are too large to fit in on-chip memory on FPGAs, running with a batch size of one. A performance analysis based on LSTM models is also presented to balance area, power, and performance in FPGA designs. When compared to state-of-the-art designs that use off-chip memory to store weights, our design achieves 1.65 times higher performance-per-watt efficiency and 1.60 times higher performance-per-DSP efficiency. When compared to CPU and GPU implementations, our hardware architecture is 23.7 and 1.3 times faster while consuming 208 and 19.2 times less energy, respectively.

Chapter 5 (Contribution 3) introduces a partially-folded hardware architecture for RNNs that maps all the layers to on-chip resources and performs computation on their own units with dedicated optimization to achieve low latency and high throughput. A novel technique for balancing IIs in multi-layer RNN inference is also presented, which improves hardware efficiency and increases design throughput. In addition, a low latency LSTM template is proposed, which enables the generation of low-latency FPGA designs with efficient resource utilization by HLS tools.

While the proposed solutions in the thesis significantly enhance the performance and efficiency of RNN designs on FPGAs, there are some limitations that merit further exploration. First, the proposed architectures are based on existing FPGA devices, leaving potential room for optimization with novel hardware resources in new FPGAs - see Section 6.2.1. Second, the focus of the thesis is on hardware-based optimizations, but numerous algorithm-based optimizations can be combined with our solutions to perform algorithm-hardware co-design, potentially yielding even higher performance - see Section 6.2.2. Furthermore, other aspects - see Section 6.2.3 - worth investigating in future work include hybrid hardware architectures, reconfigurability for RNN accelerators, and methods of improving debugging, security, and resilience of RNN accelerators. Investigating these topics could help achieve even better RNN designs on FPGAs.

In summary, this thesis presents hardware optimization approaches for RNNs on FPGAs that

address the challenges of recurrent nature and data dependencies, and low hardware utilization, and large latency. These optimizations are applied to different types of RNNs with different memory configurations, including those with weights in on-chip memory and those with weights in off-chip memory. The proposed approaches and architectures significantly improve the performance and efficiency of RNN designs on FPGAs, with speed-ups over CPU and GPU implementations and better performance compared to other FPGA designs.

6.2 Future Work

There are several areas where further research could be conducted to improve the performance and efficiency of RNN accelerators on FPGAs. Some possible directions for future work will be presented in the following sections.

6.2.1 Novel Hardware

Current and future work includes exploring the use of new FPGA resources such as the AI Engines [50] from AMD/Xilinx for RNNs. AI Engines are architected as 2D arrays consisting of multiple AI Engine tiles that are optimized for real time machine learning computation with deterministic performance. Intel also has similar engines, named AI Tensor Blocks [51] which contain dense matrix math units that are tuned for 8-bit and 4-bit integer operations with mixed precision computations. These blocks can also be ganged up to handle large vector math computations. However, these new engines are still electrical based components for computation. In the future, Optical Neural Networks (ONN) [124, 125] which performs the computation using optical elements could be a promising alternative as it provides fast data processing with low power consumption.

Besides, von Neumann architecture based accelerators suffer from large power consumption and latency due to data movement, which limits the efficiency of RNN accelerators. High-speed memory, such as on-chip SRAM, can provide high memory bandwidth but it is costly

with large silicon die area. There is also High Bandwidth Memory (HBM) [126] but its memory bandwidth is still not sufficient to meet the requirement of memory-bound applications, such as RNNs. In the future, in-memory computing [127] which runs calculations entirely in computer memory might be a promising solution to overcome the memory bottleneck in RNNs.

These studies are orthogonal to the approaches and optimizations presented in this thesis, and could be complementary to ours in order to achieve even better performance and efficiency in RNN designs.

6.2.2 Hardware/Software Co-design

In addition to exploring novel hardware components, it is also important to consider combining hardware and software optimizations in order to further improve the performance and efficiency of RNN accelerations. With the reconfigurability of FPGAs, developers can not only optimize algorithms on fixed hardware, but also customize both the algorithm and hardware together in order to achieve good performance. This design space offers many opportunities for co-optimization and adaptation as machine learning algorithms and hardware architectures evolve.

One way to extend our FPGA-based innovations to a wider range of machine learning tasks is by combining them with software optimizations, such as model compression techniques like hardware-friendly unstructured or structured pruning [70, 75], low bit-width pre-training [121] or post-training [15, 56] quantization, and neural architecture search [128]. These techniques can help to further improve the performance and efficiency of reconfigurable RNN designs, making them suitable for a wider range of applications.

6.2.3 Further Opportunities for Future Work

Other potential areas of future work for RNNs on FPGAs include:

- Investigating new hardware architectures and optimization techniques for RNNs on FPGAs, such as hybrid architectures that combine the fully-folded and partially-folded ar-

chitectures.

- Developing new tiling strategies or parallelization techniques for RNN computations on FPGAs, such as strategies that can better exploit data parallelism or improve hardware utilization.
- Evaluating the performance and energy efficiency of RNN accelerators on FPGAs in different application domains, such as natural language processing, speech recognition, and computer vision.
- Investigating new methods for mapping RNN models to single and multiple FPGAs, such as high-level synthesis tools or automatic mapping techniques that can generate efficient FPGA implementations with minimal user intervention.
- Exploring how the proposed optimizations can benefit RNN training. As the use of FPGAs for RNN inference becomes more widespread, it is possible that FPGAs could become more commonly used for RNN training as well.
- Exploring how the reconfigurability of FPGAs can benefit the RNN accelerators by having multiple designs optimized for different workloads, and selecting the one that would minimize latency and maximize performance at run time.
- Investigating methods to improve debugging, security, resilience of RNN accelerators to ensure that the accelerators are robust and can be deployed in real-world applications.

6.3 Final Thoughts

6.3.1 Transformer and RNN

The transformer [129] model uses a self-attention mechanism that allows it to process input data in parallel, rather than sequentially. This makes it much faster and more efficient than traditional recurrent neural networks (RNNs), such as LSTMs, which process input data sequentially. Transformer-based models have achieved state-of-the-art results [129, 130] and have

become the de facto standard for some tasks, such as language modeling and machine translation. An example application of large-scale language model is the famous ChatGPT [131]. Comparison between transformer and LSTM is available [130, 132, 133, 134].

Some people argue that RNNs are superseded by transformers [129], but this is not the case. Although transformers [129] have achieved promising results in large scale Natural Language Processing (NLP) tasks, RNNs are still popular in many other sequence-based tasks, delivering an acceptable level of accuracy, such as audio processing [135] and anomaly detection [26, 32]. Besides, [132] combines the components of LSTM and transformer and achieves better results than both models on their own. Furthermore, [134] shows that LSTM models can achieve better results than a transformer model on a small dataset. Transformers process all input tokens in parallel, allowing them to take advantage of parallelism but at a cost of per-layer complexity of $O(N^2)$. In contrast, RNNs have a complexity of only $O(N)$. Since it is costly to adopt transformers on long-sequence tasks, RNNs are still favorite for some domain-specific areas, such as edge computing.

To make informed decisions about when to use RNNs versus transformers, it is important to conduct a systematic evaluation of applications and workloads using benchmarks, as well as resource and performance models for CPUs, GPUs, and FPGAs. In addition, standardized evaluation metrics that capture the strengths and limitations of both architectures are necessary. These efforts would enable researchers to determine the most appropriate architecture for a given task under specific resource constraints.

Furthermore, the Block-Recurrent Transformer [136] has been proposed recently to reduce the computational cost without sacrificing accuracy, by integrating a recurrence mechanism into a transformer layer and achieving improved results over very long sentences [136]. This highlights the potential for RNNs to continue being a key component in future neural layers, resulting in more efficient neural network models. Given the high cost of running ChatGPT (millions of dollars a day [137]) and the increasing interest in making AI more efficient and accessible, integrating the proposed optimizations for RNNs in this thesis with the ideas presented in the Block-Recurrent Transformer [136] could potentially lead to more efficient and cost-effective

transformer models for use in ChatGPT [131] and other natural language processing applications. These optimizations, which focus on improving the performance and efficiency of RNNs on FPGAs, may also benefit future models with recurrence mechanisms, resulting in more efficient neural network models.

6.3.2 Design Automation

Many of the optimizations presented in this thesis are done manually, which require in-depth hardware understanding. However, this optimization process is tedious, error-prone, and must be repeated for each new application, or at least each new application domain. Besides, it becomes more challenging to create good optimized designs since compute landscape is rapidly evolving and becoming more heterogeneous with various underlying hardware units, such as GPUs, FPGAs, ASICs, CPUs, etc. The "Cambrian explosion" of novel computer architectures [138] with domain-specific hardware platforms is on the way. It becomes even more challenging when designs meet machine learning which is also a rapidly moving research field. The gap between software descriptions and optimized designs is widening.

As a result, a novel and high performance tool framework for this "Cambrian explosion" is in demand to serve as a bridge between the continuously evolving machine learning algorithms and hardware architectures. The use of automatic or machine learning guided tool frameworks could also lead to more efficient hardware/software co-optimization [139] and exploration of domain-specific and heterogeneous hardware platforms. Such frameworks could include automating hardware architecture generation or generating the code for the architecture if the architecture is programmable.

The proposed optimizations in this thesis could serve as the basis for a potential RNN design tool that automates the optimization process and enables users to customize their designs based on their specific needs and trade-offs. Fig. 6.1, which is based on Fig. 1.6, could serve as a foundation for automating diverse RNN implementations and cover end-to-end RNN design, from high-level description to implementation on various hardware platforms. In the case of a single-chip design, this figure shows the use of the partially-folded architecture introduced

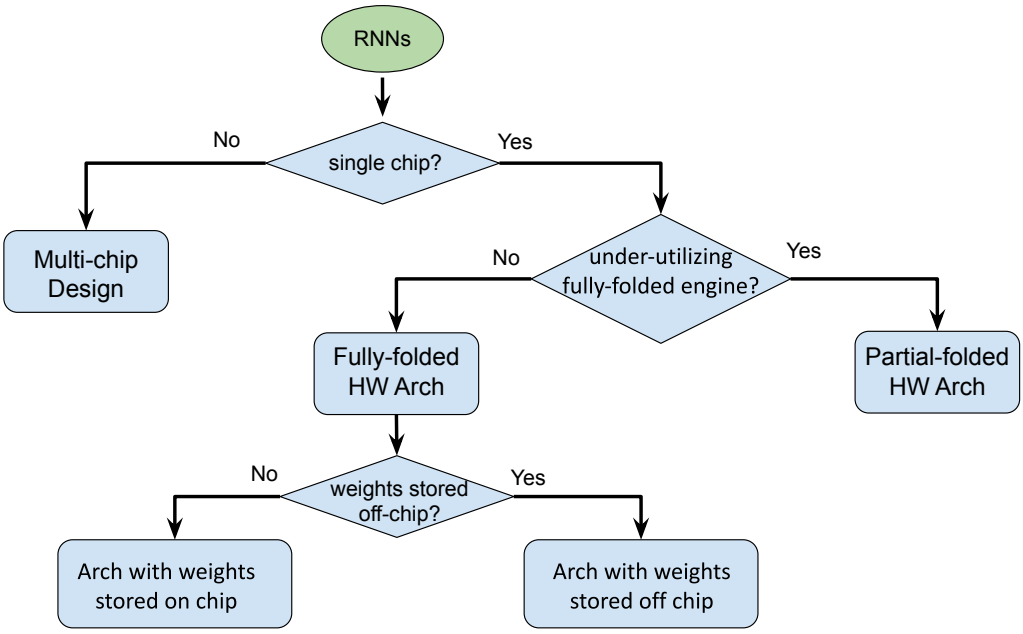


Figure 6.1: Various design options for RNNs.

in Chapter 5 when the fully-folded engine is under-utilized; otherwise, it employs the fully-folded architecture introduced in Chapters 3 and 4. The fully-folded engine is an engine with a fully-folded architecture. For multi-chip designs, it is critical to distribute the computation load across multiple chips and implement appropriate communication strategies to ensure efficient data transfer and synchronization between the chips. This could involve a combination of partially-folded and fully-folded architectures, depending on the specific requirements and constraints of the multi-chip system.

An extended version of Fig. 6.1 could involve analysis of RNN algorithm and data descriptions, prioritization of trade-offs, and multiple levels of optimization, considering both high-level and low-level optimizations. The high-level optimization would focus on algorithm optimization, such as neural architecture fine-tuning and model compression, determining the best-suited hardware platform, including CPUs, GPUs, FPGAs, or a mixture of resources, and deciding on optimal single-chip or multiple-chip, homogeneous or heterogeneous implementations. The low-level optimization would involve fine-grained tuning of the chosen architecture for the selected hardware, maximizing performance and resource utilization.

To explore the development of this extended tool architecture, initial steps could include:

1. Analyze the algorithm and data description, extracting key features from a given RNN model.

2. Prioritize trade-offs based on the user's specific needs and constraints.

3. Perform design space exploration based on the extended Fig. 6.1 to explore various configurations and optimizations for the given RNN model. This step could include evaluating the trade-offs between different RNN architectures (such as LSTM, GRU, or other variants), investigating algorithm optimizations like model compression techniques (e.g. pruning or quantization), and assessing the compatibility and performance of different hardware platforms, such as CPUs, GPUs, FPGAs, or a mixture. Determine which platform and architecture (fully-folded or partially-folded) offer better resource utilization, efficiency, and latency based on the specific needs and constraints of the user. This step automatically identifies the most suitable RNN architecture and optimization strategy for the specific application and constraints.

4. Generate an initial hardware design based on the selected RNN architecture, algorithm optimizations, and hardware platform.

5. Apply low-level optimizations tailored to the chosen hardware platform and architecture, fine-tuning the design for maximum performance and resource utilization. If the results are not satisfactory, the tool framework could return to step 3 to try different design and optimization options and iterate the process.

Developing such an end-to-end tool is an ambitious task, and the work flow summarized in Fig. 6.1 can serve as a foundation. This tool would benefit researchers and developers without hardware optimization expertise, improving their productivity and the resulting design quality which will help maximize the impact of RNN designs.

Bibliography

[1] J. Fowers, K. Ovtcharov, M. Papamichael, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, L. Adams, M. Ghandi *et al.*, "A Configurable Cloud-Scale DNN Processor for Real-Time AI," in *Proceedings of the 45th Annual International Symposium on Computer Architecture*. IEEE Press, 2018, pp. 1–14.

[2] E. Nurvitadhi, D. Kwon, A. Jafari, A. Boutros, J. Sim, P. Tomson, H. Sumbul, G. Chen, P. Knag, R. Kumar *et al.*, "Why Compete When You Can Work Together: FPGA-ASIC Integration for Persistent RNNs," in *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2019, pp. 199–207.

[3] T. Zhao, Y. Zhang, and K. Olukotun, "Serving recurrent neural networks efficiently with a spatial accelerator," *Proceedings of Machine Learning and Systems*, vol. 1, pp. 166–177, 2019.

[4] M. Auli, M. Galley, C. Quirk, and G. Zweig, "Joint language and translation modeling with recurrent neural networks," in *Proc. of EMNLP*, 2013.

[5] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using RNN encoder-decoder for statistical machine translation," *arXiv preprint arXiv:1406.1078*, 2014.

[6] Y. Goldberg, "A primer on neural network models for natural language processing," *Journal of Artificial Intelligence Research*, vol. 57, pp. 345–420, 2016.

[7] E. Arisoy, T. N. Sainath, B. Kingsbury, and B. Ramabhadran, "Deep neural network language models," in *Proceedings of the NAACL-HLT 2012 Workshop: Will We Ever*

Really Replace the N-gram Model? On the Future of Language Modeling for HLT, 2012, pp. 20–28.

[8] D. Amodei, S. Ananthanarayanan, R. Anubhai, J. Bai, E. Battenberg, C. Case, J. Casper, B. Catanzaro, Q. Cheng, G. Chen *et al.*, "Deep speech 2: End-to-end speech recognition in english and mandarin," in *International conference on machine learning*. PMLR, 2016, pp. 173–182.

[9] S. Han, J. Kang, H. Mao, Y. Hu, X. Li, Y. Li, D. Xie, H. Luo, S. Yao, Y. Wang *et al.*, "ESE: Efficient speech recognition engine with sparse LSTM on FPGA," in *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 2017, pp. 75–84.

[10] J. Yue-Hei Ng, M. Hausknecht, S. Vijayanarasimhan, O. Vinyals, R. Monga, and G. Toderici, "Beyond short snippets: Deep networks for video classification," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, pp. 4694–4702.

[11] J. Donahue, L. Anne Hendricks, S. Guadarrama, M. Rohrbach, S. Venugopalan, K. Saenko, and T. Darrell, "Long-term Recurrent Convolutional Networks for Visual Recognition and Description," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, pp. 2625–2634.

[12] N. P. Jouppi, D. H. Yoon, G. Kurian, S. Li, N. Patil, J. Laudon, C. Young, and D. Patterson, "A domain-specific supercomputer for training deep neural networks," *Communications of the ACM*, vol. 63, no. 7, pp. 67–78, 2020.

[13] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[14] V. Rybalkin, N. Wehn, M. R. Yousefi, and D. Stricker, "Hardware architecture of bidirectional long short-term memory neural network for optical character recognition," in *Proceedings of the Conference on Design, Automation & Test in Europe*, 2017, pp. 1394–1399.

[15] V. Rybalkin, A. Pappalardo, M. M. Ghaffar, G. Gambardella, N. Wehn, and M. Blott, "FINN-L: Library extensions and design trade-off analysis for variable precision LSTM networks on FPGAs," in *28th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2018.

[16] V. Rybalkin, C. Sudarshan, C. Weis, J. Lappas, N. Wehn, and L. Cheng, "Efficient hardware architectures for 1D-and MD-LSTM networks," *Journal of Signal Processing Systems*, vol. 92, no. 11, pp. 1219–1245, 2020.

[17] M. Zhang, S. Rajbhandari, W. Wang, and Y. He, "DeepCPU: Serving RNN-based deep learning models 10x faster," in *Proc. of USENIX ATC*, 2018, pp. 951–965.

[18] A. Ardakani, Z. Ji, and W. J. Gross, "Learning to Skip Ineffectual Recurrent Computations in LSTMs," *arXiv preprint arXiv:1811.10396*, 2018.

[19] Y. Guan, H. Liang, N. Xu, W. Wang, S. Shi, X. Chen, G. Sun, W. Zhang, and J. Cong, "FP-DNN: An automated framework for mapping deep neural networks onto FPGAs with RTL-HLS hybrid templates," in *Field-Programmable Custom Computing Machines (FCCM), 2017 IEEE 25th Annual International Symposium on*. IEEE, 2017, pp. 152–159.

[20] P. Gao, L. Yu, Y. Wu, and J. Li, "Low latency RNN inference with cellular batching," in *Proceedings of the Thirteenth EuroSys Conference*, 2018, pp. 1–15.

[21] F. Silfa, J. M. Arnau, and A. Gonzalez, "E-BATCH: Energy-Efficient and High-Throughput RNN Batching," *arXiv preprint arXiv:2009.10656*, 2020.

[22] A. Boutros, E. Nurvitadhi, R. Ma, S. Gribok, Z. Zhao, J. C. Hoe, V. Betz, and M. Langhammer, "Beyond peak performance: Comparing the real performance of AI-optimized FPGAs and GPUs," in *2020 International Conference on Field-Programmable Technology (ICFPT)*. IEEE, 2020, pp. 10–19.

[23] Y. LeCun, "Deep Learning Hardware: Past, Present, and Future," in *2019 IEEE International Solid-State Circuits Conference-(ISSCC)*. IEEE, 2019, pp. 12–19.

[24] Z. Que, H. Nakahara, E. Nurvitadhi, H. Fan, C. Zeng, J. Meng, X. Niu, and W. Luk, "Optimizing Reconfigurable Recurrent Neural Networks," in *IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2020, pp. 10–18.

[25] L. Peng, W. Shi, J. Zhang, and S. Irving, "Exploiting Model-Level Parallelism in Recurrent Neural Network Accelerators," in *2019 IEEE 13th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC)*. IEEE, 2019, pp. 241–248.

[26] E. A. Moreno, B. Borzyszkowski, M. Pierini, J.-R. Vlimant, and M. Spiropulu, "Source-agnostic gravitational-wave detection with recurrent autoencoders," *Machine Learning: Science and Technology*, 2022.

[27] A. Hannun, C. Case, J. Casper, B. Catanzaro, G. Diamos, E. Elsen, R. Prenger, S. Satheesh, S. Sengupta, A. Coates *et al.*, "Deep speech: Scaling up end-to-end speech recognition," *arXiv preprint arXiv:1412.5567*, 2014.

[28] J. Duato, S. Yalamanchili, and L. Ni, *Interconnection networks*. Morgan Kaufmann, 2003.

[29] Z. Que, H. Nakahara, E. Nurvitadhi, A. Boutros, H. Fan, C. Zeng, J. Meng, K. H. Tsoi, X. Niu, and W. Luk, "Recurrent Neural Networks With Column-Wise Matrix-Vector Multiplication on FPGAs," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 30, no. 2, pp. 227–237, 2022.

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

[31] Z. Que, Y. Zhu, H. Fan, J. Meng, X. Niu, and W. Luk, "Mapping Large LSTMs to FPGAs with Weight Reuse," *Journal of Signal Processing Systems*, vol. 92, no. 9, pp. 965–979, 2020.

[32] Z. Que, E. Wang, U. Marikar, E. Moreno, J. Ngadiuba, H. Javed, B. Borzyszkowski, T. Aarrestad, V. Loncar, S. Summers *et al.*, "Accelerating recurrent neural networks

for gravitational wave experiments," in *2021 IEEE 32nd International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE, 2021, pp. 117–124.

[33] M. Ferianc, Z. Que, H. Fan, W. Luk, and M. Rodrigues, "Optimizing Bayesian Recurrent Neural Networks on an FPGA-based Accelerator," in *2021 International Conference on Field-Programmable Technology (ICFPT)*. IEEE, 2021, pp. 1–10.

[34] Z. Que, M. Loo, H. Fan, M. Pierini, A. Tapper, and W. Luk, "Optimizing Graph Neural Networks for Jet Tagging in Particle Physics on FPGAs," in *Field Programmable Logic and Applications (FPL), 32th International Conference on*. IEEE, 2022.

[35] Z. Que, M. Loo, H. Fan, M. Blott, M. Pierini, A. D. Tapper, and W. Luk, "LL-GNN: Low Latency Graph Neural Networks on FPGAs for Particle Detectors," *arXiv preprint arXiv:2209.14065*, 2022.

[36] P. J. Werbos, "Backpropagation through time: what it does and how to do it," *Proceedings of the IEEE*, vol. 78, no. 10, pp. 1550–1560, 1990.

[37] Y. Bengio, P. Simard, and P. Frasconi, "Learning long-term dependencies with gradient descent is difficult," *IEEE transactions on neural networks*, vol. 5, no. 2, pp. 157–166, 1994.

[38] Y. Guan, Z. Yuan, G. Sun, and J. Cong, "FPGA-based accelerator for long short-term memory recurrent neural networks," in *Design Automation Conference (ASP-DAC), 2017 22nd Asia and South Pacific*. IEEE, 2017, pp. 629–634.

[39] K. Cho, B. van Merriënboer, D. Bahdanau, and Y. Bengio, "On the Properties of Neural Machine Translation: Encoder–Decoder Approaches," *Syntax, Semantics and Structure in Statistical Translation*, p. 103, 2014.

[40] S. Mangal, P. Joshi, and R. Modak, "LSTM vs. GRU vs. Bidirectional RNN for script generation," *arXiv preprint arXiv:1908.04332*, 2019.

[41] P. T. Yamak, L. Yujian, and P. K. Gadosey, "A comparison between ARIMA, LSTM, and GRU for time series forecasting," in *Proceedings of the 2019 2nd International Conference on Algorithms, Computing and Artificial Intelligence*, 2019, pp. 49–55.

[42] S. Yang, X. Yu, and Y. Zhou, "LSTM and GRU neural network performance comparison study: Taking Yelp review dataset as an example," in *2020 International workshop on electronic communication and artificial intelligence (IWECAI)*. IEEE, 2020, pp. 98–101.

[43] M. R. Raza, W. Hussain, and J. M. Merigó, "Cloud sentiment accuracy comparison using RNN, LSTM and GRU," in *2021 Innovations in intelligent systems and applications conference (ASYU)*. IEEE, 2021, pp. 1–5.

[44] R. Zhang, Z. Guo, Y. Meng, S. Wang, S. Li, R. Niu, Y. Wang, Q. Guo, and Y. Li, "Comparison of ARIMA and LSTM in Forecasting the Incidence of HFMD Combined and Uncombined with Exogenous Meteorological Variables in Ningbo, China," *International journal of environmental research and public health*, vol. 18, no. 11, p. 6174, 2021.

[45] S. Siami-Namini, N. Tavakoli, and A. S. Namin, "A comparison of ARIMA and LSTM in forecasting time series," in *2018 17th IEEE international conference on machine learning and applications (ICMLA)*. IEEE, 2018, pp. 1394–1401.

[46] "Intel Arria Series FPGAs and SoCs ," https://www.intel.co.uk/content/www/uk/en/products/details/fpga/arria.html, 2023.

[47] "Zynq 7000 SoC," https://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html, 2023.

[48] Intel, "Intel Extends its Multi-Platform RISC-V Support by Partnering with Prominent RISC-V Tools Provider, Ashling," in *White Paper 01317*. Intel.

[49] "MicroBlaze Soft Processor Core," https://www.xilinx.com/products/design-tools/microblaze.html, 2023.

[50] "Xilinx AI Engines and Their Applications," in *WP506(v1.1)*, July 10, 2020.

[51] M. Langhammer, E. Nurvitadhi, B. Pasca, and S. Gribok, "Stratix 10 NX Architecture and Applications," in *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2021, pp. 57–67.

[52] "Intel Stratix Series FPGA and SoC FPGA," https://www.intel.co.uk/content/www/uk/en/products/details/fpga/stratix.html, 2023.

[53] "Intel Agilex FPGA," https://www.intel.co.uk/content/www/uk/en/products/details/fpga/agilex.html, 2023.

[54] S. Wang, Z. Li, C. Ding, B. Yuan, Q. Qiu, Y. Wang, and Y. Liang, "C-LSTM: Enabling Efficient LSTM using Structured Compression Techniques on FPGAs," in *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 2018, pp. 11–20.

[55] X. Zhang, W. Jiang, and J. Hu, "Achieving full parallelism in LSTM via a unified accelerator design," in *2020 IEEE 38th International Conference on Computer Design (ICCD)*. IEEE, 2020, pp. 469–477.

[56] V. Rybalkin and N. Wehn, "When Massive GPU Parallelism Ain't Enough: A Novel Hardware Architecture of 2D-LSTM Neural Network," in *The ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2020.

[57] J. Liu, J. Wang, Y. Zhou, and F. Liu, "A cloud server oriented FPGA accelerator for LSTM recurrent neural network," *IEEE Access*, vol. 7, pp. 122 408–122 418, 2019.

[58] R. Yazdani, O. Ruwase, M. Zhang, Y. He, J.-M. Arnau, and A. González, "SHARP: An Adaptable, Energy-Efficient Accelerator for Recurrent Neural Networks," *ACM Transactions on Embedded Computing Systems (TECS)*, 2022.

[59] A. X. M. Chang, B. Martini, and E. Culurciello, "Recurrent neural networks hardware implementation on FPGA," *arXiv preprint arXiv:1511.05552*, 2015.

[60] N. Park, Y. Kim, D. Ahn, T. Kim, and J.-J. Kim, "Time-step interleaved weight reuse

for LSTM neural network computing," in *Proceedings of the ACM/IEEE International Symposium on Low Power Electronics and Design*, 2020, pp. 13–18.

[61] S. Tewari, A. Kumar, and K. Paul, "SACC: split and combine approach to reduce the off-chip memory accesses of LSTM accelerators," in *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2022, pp. 580–583.

[62] Z. Que, Y. Liu, C. Guo, X. Niu, Y. Zhu, and W. Luk, "Real-time Anomaly Detection for Flight Testing using AutoEncoder and LSTM," in *2019 International Conference on Field-Programmable Technology (ICFPT)*. IEEE, 2019, pp. 379–382.

[63] E. E. Khoda, D. Rankin, R. T. de Lima, P. Harris, S. Hauck, S.-C. Hsu, M. Kagan, V. Loncar, C. Paikara, R. Rao *et al.*, "Ultra-low latency recurrent neural network inference on FPGAs for physics applications with hls4ml," *arXiv preprint arXiv:2207.00559*, 2022.

[64] Y. Sun, A. Ben Ahmed, and H. Amano, "Acceleration of deep recurrent neural networks with an FPGA cluster," in *Proceedings of the 10th International Symposium on Highly-Efficient Accelerators and Reconfigurable Technologies*, 2019, pp. 1–4.

[65] Y. Sun and H. Amano, "FiC-RNN: A multi-FPGA acceleration framework for deep recurrent neural networks," *IEICE Transactions on Information and Systems*, vol. 103, no. 12, pp. 2457–2462, 2020.

[66] D. Kwon, S. Hur, H. Jang, E. Nurvitadhi, and J. Kim, "Scalable Multi-FPGA Acceleration for Large RNNs with Full Parallelism Levels," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2020, pp. 1–6.

[67] Z. Chen, A. Howe, H. T. Blair, and J. Cong, "CLINK: Compact LSTM inference kernel for energy efficient neurofeedback devices," in *Proceedings of the International Symposium on Low Power Electronics and Design*, 2018, pp. 1–6.

[68] C. Gao, D. Neil, E. Ceolini, S.-C. Liu, and T. Delbruck, "DeltaRNN: A power-efficient recurrent neural network accelerator," in *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 2018, pp. 21–30.

[69] J. Wu, F. Li, Z. Chen, and X. Xiang, "A 3.89-GOPS/mW Scalable Recurrent Neural Network Processor With Improved Efficiency on Memory and Computation," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 27, no. 12, pp. 2939–2943, 2019.

[70] S. Cao, C. Zhang, Z. Yao, W. Xiao, L. Nie, D. Zhan, Y. Liu, M. Wu, and L. Zhang, "Efficient and Effective Sparse LSTM on FPGA with Bank-Balanced Sparsity," in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 2019, pp. 63–72.

[71] R. Shi, J. Liu, K.-H. H. So, S. Wang, and Y. Liang, "E-LSTM: Efficient inference of sparse LSTM on embedded heterogeneous system," in *2019 56th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2019, pp. 1–6.

[72] G. Nan, C. Wang, W. Liu, and F. Lombardi, "DC-LSTM: Deep Compressed LSTM with Low Bit-Width and Structured Matrices," in *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, 2020, pp. 1–5.

[73] Z. Chen, G. J. Blair, H. T. Blair, and J. Cong, "BLINK: bit-sparse LSTM inference kernel enabling efficient calcium trace extraction for neurofeedback devices," in *Proceedings of the ACM/IEEE International Symposium on Low Power Electronics and Design*, 2020, pp. 217–222.

[74] Y. Zheng, H. Yang, Y. Jia, and Z. Huang, "PermLSTM: A High Energy-Efficiency LSTM Accelerator Architecture," *Electronics*, vol. 10, no. 8, p. 882, 2021.

[75] C. Gao, T. Delbruck, and S.-C. Liu, "Spartus: A 9.4 top/s FPGA-based LSTM accelerator exploiting spatio-temporal sparsity," *IEEE Transactions on Neural Networks and Learning Systems*, 2022.

[76] J. Jiang, T. Xiao, J. Xu, D. Wen, L. Gao, and Y. Dou, "A low-latency LSTM accelerator using balanced sparsity based on FPGA," *Microprocessors and Microsystems*, vol. 89, p. 104417, 2022.

[77] Z. Chen, H. T. Blair, and J. Cong, "Energy Efficient LSTM Inference Accelerator for Real-Time Causal Prediction," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 2022.

[78] R. Prabhakar, Y. Zhang, D. Koeplinger, M. Feldman, T. Zhao, S. Hadjis, A. Pedram, C. Kozyrakis, and K. Olukotun, "Plasticine: A reconfigurable architecture for parallel patterns," in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2017, pp. 389–402.

[79] J. Kim, J. Kim, and T.-H. Kim, "AERO: A 1.28 MOP/s/LUT Reconfigurable Inference Processor for Recurrent Neural Networks in a Resource-Limited FPGA," *Electronics*, vol. 10, no. 11, p. 1249, 2021.

[80] Z. Que, H. Nakahara, H. Fan, J. Meng, K. H. Tsoi, X. Niu, E. Nurvitadhi, and W. Luk, "A Reconfigurable Multithreaded Accelerator for Recurrent Neural Networks," in *2020 International Conference on Field-Programmable Technology (ICFPT)*. IEEE, 2020, pp. 20–28.

[81] Z. Que, H. Nakahara, H. Fan, H. Li, J. Meng, K. H. Tsoi, X. Niu, E. Nurvitadhi, and W. Luk, "Remarn: A Reconfigurable Multi-threaded Multi-core Accelerator for Recurrent Neural Networks," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 16, no. 1, 2022.

[82] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems," 2015, software available from tensorflow.org. [Online]. Available: https://www.tensorflow.org/

[83] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin,

N. Gimelshein, L. Antiga *et al.*, "PyTorch: An imperative style, high-performance deep learning library," *Advances in neural information processing systems*, vol. 32, 2019.

[84] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, 2017, pp. 1–12.

[85] D. Abts, J. Ross, J. Sparling, M. Wong-VanHaren, M. Baker, T. Hawkins, A. Bell, J. Thompson, T. Kahsai, G. Kimmell *et al.*, "Think fast: A tensor streaming processor (TSP) for accelerating deep learning workloads," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2020, pp. 145–158.

[86] D. Abts, J. Kim, G. Kimmell, M. Boyd, K. Kang, S. Parmar, A. Ling, A. Bitar, I. Ahmed, and J. Ross, "The Groq Software-defined Scale-out Tensor Streaming Multiprocessor: From chips-to-systems architectural overview," in *2022 IEEE Hot Chips 34 Symposium (HCS)*. IEEE Computer Society, 2022, pp. 1–69.

[87] J. Fowers, "Hold my FPGA: GroqChip takes the Cake for LSTM," https://groq.com/hold-my-fpga-groqchip-takes-the-cake-for-lstm/, 2023.

[88] R. Prabhakar and S. Jairath, "SambaNova SN10 RDU: Accelerating software 2.0 with dataflow," in *2021 IEEE Hot Chips 33 Symposium (HCS)*. IEEE, 2021, pp. 1–37.

[89] SambaNova, "Ultra-fast Recurrent Neural Networks with SambaNova's Reconfigurable Dataflow Architecture," https://sambanova.ai/blog/ultra-fast-recurrent-neural-networks-with-sambaNovas-reconfigurable-dataflow-architecture/, 2023.

[90] Z. Sun, Y. Zhu, Y. Zheng, H. Wu, Z. Cao, P. Xiong, J. Hou, T. Huang, and Z. Que, "FPGA acceleration of LSTM based on data for test flight," in *2018 IEEE International Conference on Smart Cloud (SmartCloud)*. IEEE, 2018, pp. 1–6.

[91] W. Zaremba, I. Sutskever, and O. Vinyals, "Recurrent neural network regularization," *arXiv preprint arXiv:1409.2329*, 2014.

[92] S. Pal, J. Beaumont, D.-H. Park, A. Amarnath, S. Feng, C. Chakrabarti, H.-S. Kim, D. Blaauw, T. Mudge, and R. Dreslinski, "Outerspace: An outer product based sparse matrix multiplication accelerator," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2018, pp. 724–736.

[93] Z. Zhang, H. Wang, S. Han, and W. J. Dally, "SpArch: Efficient architecture for sparse matrix multiplication," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2020, pp. 261–274.

[94] E. Nurvitadhi, A. Mishra, Y. Wang, G. Venkatesh, and D. Marr, "Hardware accelerator for analytics of sparse data," in *Proceedings of the 2016 Conference on Design, Automation & Test in Europe*. EDA Consortium, 2016, pp. 1616–1621.

[95] Xilinx, "Deep Learning with INT8 Optimization on Xilinx Devices," 2017. [Online]. Available: https://www.xilinx.com/support/documentation/whitepapers/wp486-deep-learning-int8.pdf

[96] M. Langhammer, B. Pasca, G. Baeckler, and S. Gribok, "Extracting INT8 Multipliers from INT18 Multipliers," in *International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2019.

[97] A. Boutros, S. Yazdanshenas, and V. Betz, "Embracing diversity: Enhanced DSP blocks for low-precision deep learning on FPGAs," in *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2018, pp. 35–357.

[98] Intel, "Intel Agilex Variable Precision DSP Blocks User Guide," 2020.

[99] E. Bank-Tavakoli, S. A. Ghasemzadeh, M. Kamal, A. Afzali-Kusha, and M. Pedram, "Polar: A pipelined/overlapped FPGA-based LSTM accelerator," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 28, no. 3, pp. 838–842, 2019.

[100] J. C. Ferreira and J. Fonseca, "An FPGA implementation of a long short-term memory neural network," in *International Conference on ReConFigurable Computing and FPGAs (ReConFig),*. IEEE, 2016.

[101] H. Fan, H.-C. Ng, S. Liu, Z. Que, X. Niu, and W. Luk, "Reconfigurable Acceleration of 3D-CNNs for Human Action Recognition with Block Floating-Point Representation," 2018.

[102] R. Zhao, X. Niu, Y. Wu, W. Luk, and Q. Liu, "Optimizing CNN-based object detection algorithms on embedded FPGA platforms," in *International Symposium on Applied Reconfigurable Computing*. Springer, 2017, pp. 255–267.

[103] T. Cooijmans, N. Ballas, C. Laurent, Ç. Gülçehre, and A. Courville, "Recurrent batch normalization," *arXiv preprint arXiv:1603.09025*, 2016.

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

[105] X. Zhang, J. Wang, C. Zhu, Y. Lin, J. Xiong, W.-m. Hwu, and D. Chen, "DNNBuilder: an automated tool for building high-performance DNN hardware accelerators for FPGAs," in *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2018, pp. 1–8.

[106] H. Nakahara, Z. Que, and W. Luk, "High-Throughput Convolutional Neural Network on an FPGA by Customized JPEG Compression," in *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2020, pp. 1–9.

[107] Xilinx, "SDSoC Profiling and Optimization Guide."

[108] J. de Fine Licht, M. Besta, S. Meierhans, and T. Hoefler, "Transformations of High-Level Synthesis Codes for High-Performance Computing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 5, pp. 1014–1029, 2020.

[109] J. Duarte, S. Han, P. Harris, S. Jindariani, E. Kreinar, B. Kreis, J. Ngadiuba, M. Pierini, R. Rivera, N. Tran *et al.*, "Fast inference of deep neural networks in FPGAs for particle physics," *Journal of Instrumentation*, vol. 13, no. 07, p. P07027, 2018.

[110] F. Wojcicki, Z. Que, A. D. Tapper, and W. Luk, "Accelerating Transformer Neural Networks on FPGAs for High Energy Physics Experiments," in *2022 International Conference on Field-Programmable Technology (ICFPT)*. IEEE, 2022, pp. 1–8.

[111] D. Thorwarth and D. A. Low, "Technical Challenges of Real-Time Adaptive MR-Guided Radiotherapy," *Frontiers in Oncology*, vol. 11, p. 332, 2021.

[112] S. Denholm, H. Inoue, T. Takenaka, T. Becker, and W. Luk, "Low latency FPGA acceleration of market data feed arbitration," in *2014 IEEE 25th International Conference on Application-Specific Systems, Architectures and Processors*. IEEE, 2014, pp. 36–40.

[113] Z. Que, M. Loo, and W. Luk, "Reconfigurable Acceleration of Graph Neural Networks for Jet Identification in Particle Physics," in *2022 IEEE 4rd International Conference on Artificial Intelligence Circuits and Systems (AICAS)*. IEEE, 2022.

[114] E. Moreno. [Online]. Available: https://github.com/eric-moreno/Anomaly-Detection-Autoencoder

[115] Y. Shen, M. Ferdman, and P. Milder, "Maximizing CNN accelerator efficiency through resource partitioning," in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2017, pp. 535–547.

[116] X. Zhang, H. Ye, J. Wang, Y. Lin, J. Xiong, W.-m. Hwu, and D. Chen, "DNNExplorer: a framework for modeling and exploring a novel paradigm of FPGA-based DNN accelerator," in *Proceedings of the 39th International Conference on Computer-Aided Design*, 2020, pp. 1–9.

[117] B. Moons, K. Goetschalckx, N. Van Berckelaer, and M. Verhelst, "Minimum energy quantized neural networks," in *2017 51st Asilomar Conference on Signals, Systems, and Computers*. IEEE, 2017, pp. 1921–1925.

[118] "Generate Gravitational-Wave Data (GGWD)," https://github.com/timothygebhard/ggwd, 2019.

[119] A. Nitz, I. Harry, D. Brown, C. M. Biwer, J. Willis, T. D. Canton, C. Capano, L. Pekowsky, T. Dent, A. R. Williamson, G. S. Davies, S. De, M. Cabero, B. Machenschalk, P. Kumar, S. Reyes, D. Macleod, F. Pannarale, dfinstad, T. Massinger, M. Tápai, L. Singer, S. Khan, S. Fairhurst, S. Kumar, A. Nielsen, SSingh087, shasvath, I. Dorrington, and B. U. V. Gadre, "gwastro/pycbc: PyCBC release v1.16.9," Aug. 2020. [Online]. Available: https://doi.org/10.5281/zenodo.3993665

[120] LIGO Scientific Collaboration, "LIGO Algorithm Library - LALSuite," free software (GPL), 2018.

[121] C. N. Coelho, A. Kuusela, S. Li, H. Zhuang, J. Ngadiuba, T. K. Aarrestad, V. Loncar, M. Pierini, A. A. Pol, and S. Summers, "Automatic heterogeneous quantization of deep neural networks for low-latency inference on the edge for particle detectors," *Nature Machine Intelligence*, vol. 3, no. 8, pp. 675–686, 2021.

[122] R. Rao, *Implementation of Long Short-Term Memory Neural Networks in High-Level Synthesis Targeting FPGAs.* Master's Thesis, University of Washington, 2020.

[123] Y. H. Lee, D. J. Moss, J. Faraone, P. Blackmore, D. Salmond, D. Boland, P. H. Leong *et al.*, "Long Short-Term Memory for Radio Frequency Spectral Prediction and its Real-Time FPGA Implementation," in *MILCOM 2018-2018 IEEE Military Communications Conference (MILCOM).* IEEE, 2018, pp. 1–9.

[124] H. Zhang, M. Gu, X. Jiang, J. Thompson, H. Cai, S. Paesani, R. Santagati, A. Laing, Y. Zhang, M. Yung *et al.*, "An optical neural chip for implementing complex-valued neural network," *Nature communications*, vol. 12, no. 1, pp. 1–11, 2021.

[125] T. Wang, S.-Y. Ma, L. G. Wright, T. Onodera, B. C. Richard, and P. L. McMahon, "An optical neural network using less than 1 photon per multiplication," *Nature Communications*, vol. 13, no. 1, pp. 1–8, 2022.

[126] S. JEDEC, "High Bandwidth Memory (HBM) DRAM," *JESD235*, pp. 0018–9340, 2013.

[127] A. Sebastian, M. Le Gallo, R. Khaddam-Aljameh, and E. Eleftheriou, "Memory devices

and applications for in-memory computing," *Nature nanotechnology*, vol. 15, no. 7, pp. 529–544, 2020.

[128] B. Zoph and Q. V. Le, "Neural architecture search with reinforcement learning," *arXiv preprint arXiv:1611.01578*, 2016.

[129] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[130] S. Lakew, M. Cettolo, and M. Federico, "A Comparison of Transformer and Recurrent Neural Networks on Multilingual Neural Machine Translation," in *Proceedings of the 27th International Conference on Computational Linguistics (COLING)*, 2018, pp. 641–652.

[131] OpenAI, "ChatGPT [Large language model]. ," *https://chat.openai.com/chat*, 2023.

[132] M. X. Chen, O. Firat, A. Bapna, M. Johnson, W. Macherey, G. Foster, L. Jones, M. Schuster, N. Shazeer, N. Parmar *et al.*, "The Best of Both Worlds: Combining Recent Advances in Neural Machine Translation," in *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2018, pp. 76–86.

[133] S. Karita, N. Chen, T. Hayashi, T. Hori, H. Inaguma, Z. Jiang, M. Someki, N. E. Y. Soplin, R. Yamamoto, X. Wang *et al.*, "A comparative study on transformer vs RNN in speech applications," in *2019 IEEE Automatic Speech Recognition and Understanding Workshop (ASRU)*. IEEE, 2019, pp. 449–456.

[134] A. Ezen-Can, "A Comparison of LSTM and BERT for Small Corpus," *arXiv preprint arXiv:2009.05451*, 2020.

[135] J. S. P. Giraldo, S. Lauwereins, K. Badami, and M. Verhelst, "Vocell: A 65-nm Speech-Triggered Wake-Up SoC for 10-uW Keyword Spotting and Speaker Verification," *IEEE Journal of Solid-State Circuits*, vol. 55, no. 4, pp. 868–878, 2020.

[136] D. Hutchins, I. Schlag, Y. Wu, E. Dyer, and B. Neyshabur, "Block-recurrent transformers," *arXiv preprint arXiv:2203.07852*, 2022.

[137] J. Koetsier, "ChatGPT Burns Millions Every Day. Can Computer Scientists Make AI One Million Times More Efficient?" https://www.forbes.com/sites/johnkoetsier/2023/02/10/chatgpt-burns-millions-every-day-can-computer-scientists-make-ai-one-million-times-more-efficient/ forbes.com, 2023.

[138] J. L. Hennessy and D. A. Patterson, "A new golden age for computer architecture," *Communications of the ACM*, vol. 62, no. 2, pp. 48–60, 2019.

[139] Z. Que, S. Liu, M. Rognlien, C. Guo, D. F. C. G, and W. Luk, "MetaML: Automating Customizable Cross-Stage Design-Flow for Deep Learning Acceleration," in *2023 33th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2023.