# OPTIMIZING NEURAL NETWORKS PERFORMANCE ON PARALLEL ARCHITECTURES

**SAURABH TEWARI**


DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY DELHI

JULY 2023

# OPTIMIZING NEURAL NETWORKS PERFORMANCE ON PARALLEL ARCHITECTURES

by

SAURABH TEWARI

Department of Computer Science and Engineering

Submitted

in fulfillment of the requirements of the degree of Doctor of Philosophy

to the

**INDIAN INSTITUTE OF TECHNOLOGY DELHI**

**JULY 2023**

# Certificate

This is to certify that the thesis titled "**OPTIMIZING NEURAL NETWORKS PERFORMANCE ON PARALLEL ARCHITECTURES**" being submitted by **Saurabh Tewari** for the award of **Doctor of Philosophy** in Computer Science and Engineering is a record of bonafide work carried out by him under my guidance and supervision in the Department of Computer Science and Engineering, Indian Institute of Technology Delhi. The work presented in this thesis has not been submitted elsewhere, either in part or full, for the award of any other degree or diploma unless otherwise stated explicitly.

The works presented in Chapter 5 on Self Organizing Maps (SOMs) were conducted in collaboration with students at KTH Royal Institute of Technology, Sweden. We acknowledge their valuable contributions explicitly in the corresponding chapter.

**Anshul Kumar**
Professor
Department of Computer Science and Engg.
Indian Institute of Technology Delhi
New Delhi- 110016

**Kolin Paul**
Professor
Department of Computer Science and Engg.
Indian Institute of Technology Delhi
New Delhi- 110016

# Acknowledgments

I am deeply grateful to my esteemed Ph.D. supervisors, Prof. Anshul Kumar and Prof. Kolin Paul, for their exceptional guidance and unwavering support throughout my research journey. Their technical expertise and invaluable insights have been instrumental in shaping my academic growth and thesis work. They have provided mentorship and valuable advice on various aspects of academic life, proving to be excellent mentors in every sense. Their constant availability for discussions, constructive suggestions, and motivation to complete my thesis on time have been indispensable. During the challenging times of the COVID-19 lockdown, their unwavering support kept me focused and encouraged.

Heartfelt appreciation goes to my research committee members, Prof. M. Balakrishnan and Prof. Preeti Ranjan Panda, for their valuable suggestions and feedback on my thesis work, enriching its quality and depth. I am also sincerely grateful to Prof. Ahmed Hemani, a professor at KTH Sweden, for allowing me to work in their state-of-the-art CGRA accelerator labs, significantly enhancing my research exposure and knowledge.

I extend my gratitude to all the instructors at IIT Delhi who imparted valuable skills during my coursework, contributing significantly to my overall academic growth.

Special thanks go to my friends at IIT Delhi, Rajesh Kedia, Sandeep Kumar, Dishant Goyal, and Omais Shafi, for their unwavering support and valuable suggestions on useful writing tools, aiding me in crafting research papers and creating graphical representations. In particular, I want to acknowledge Rajesh Kedia for his meticulous review and invaluable feedback. I am grateful to Vandana Ahluwalia, the lab in-charge, for providing access to the labs and resources and to Som Dutt Sharma from the DHD-LAB for granting access to the FPGA boards and essential software, which played a critical role in my research.

I am deeply thankful to my family members, including my wife, parents, and daughters, whose unwavering support and understanding made this achievement possible. Their love, encouragement, and sacrifices have been a constant source of motivation and strength. Lastly,

I express gratitude to the Almighty for giving me the strength and opportunity to work with wonderful and helpful people.

I also want to acknowledge all those whose names may not be mentioned here but have contributed in various ways, big or small, to my personal and academic growth. Their collective support and encouragement have been the driving force behind the successful completion of my Ph.D. thesis, and I humbly acknowledge their invaluable contributions.

**Saurabh Tewari**

# Abstract

In recent years, Neural Networks (NNs) have experienced tremendous growth in applications across diverse fields, revolutionizing industries such as healthcare, agriculture, surveillance, and autonomous systems. The availability of large datasets and powerful computing systems, coupled with user-friendly libraries like Tensorflow and PyTorch, has enabled NNs to achieve human-like performance. As a result, NN-based applications have become an integral part of our daily lives, from face recognition and chatbots to shopping recommendations and medical diagnoses.

This Ph.D. thesis focuses on optimizing the inferencing phase of NNs for edge devices with limited computing resources and tight energy budgets. Edge devices, such as smartphones and wearable devices, are now widely equipped with NN applications, necessitating energy-efficient solutions for improved user experiences and privacy protection. The study encompasses three classes of NNs: Self Organizing Maps (SOMs), Convolutional Neural Networks (CNNs), and Recurrent Neural Networks (RNNs), each posing unique challenges in terms of data reuse and energy efficiency.

The thesis introduces the neural network model, illustrating the structure and different types of NNs, including feedforward, recurrent, and deep neural networks. The development and deployment phases of NN applications are also explained, emphasizing the significance of optimizing the inferencing phase for edge devices.

In the context of CNNs, the research focuses on optimizing data reuse through partitioning and scheduling schemes. An analytical framework is developed to estimate off-chip memory

accesses, considering architectural constraints and data shape. The framework enables the comparison of various partitioning and scheduling approaches, facilitating the discovery of optimal solutions to improve energy efficiency and throughput for different CNN layers.

For RNNs, which present additional challenges due to their dependency on previous time-step computations, a novel data reuse approach is proposed. This approach efficiently reuses weights of large matrices for consecutive time steps, irrespective of on-chip memory size. FPGA implementations of Long-Short Term Memory Network (LSTM) accelerators demonstrate the effectiveness of this approach in enhancing energy efficiency and throughput for popular LSTM models.

For SOMs, the study explores the impact of quantization techniques on accuracy and energy efficiency. A custom semi-systolic array design is devised, and a trade-off between NN accuracy and energy consumption is analyzed for different bit-width implementations. The study analyzes the benefits and trade-offs of using different bit resolutions for SOMs, catering to energy-constrained systems where area, power, and performance are critical considerations.

In conclusion, this Ph.D. thesis advances the state-of-the-art energy-efficient acceleration of modern NNs for edge devices. By proposing novel data reuse techniques and analytical frameworks, the research provides valuable insights for improving the inferencing phase, thus contributing to the widespread adoption of NN-based applications in energy-constrained environments. The work also opens new avenues for future research, pushing the boundaries of NN optimization and edge AI applications.

# सार

हाल के वर्षों में, न्यूरल नेटवर्क्स (एनएन) ने विभिन्न क्षेत्रों में अनुप्रयोगों में जबरदस्त वृद्धि का अनुभव किया है, जिससे स्वास्थ्य देखभाल, कृषि, निगरानी और स्वायत्त प्रणालियों जैसे उद्योगों में क्रांति आ गई है। बड़े डेटासेट और शक्तिशाली कंप्यूटिंग सिस्टम की उपलब्धता, साथ में टेन्सरफ़्लो और पायटोरच जैसी उपयोगकर्ता-अनुकूल लाइब्रेरीज़ ने एनएन को मानव-जैसा प्रदर्शन प्राप्त करने में सक्षम बनाया है। परिणामस्वरूप, एनएन-आधारित एप्लिकेशन चेहरे की पहचान और चैटबॉट से लेकर शॉपिंग अनुशंसाओं और चिकित्सा निदान तक हमारे दैनिक जीवन का एक अभिन्न अंग बन गए हैं।

यह पीएच.डी. थीसिस सीमित कंप्यूटिंग संसाधनों और तंग ऊर्जा बजट वाले एज उपकरणों के लिए न्यूरल नेटवर्क (एनएन) के अनुमान चरण को अनुकूलित करने पर केंद्रित है। स्मार्टफ़ोन और पहनने योग्य डिवाइस जैसे एज डिवाइस अब व्यापक रूप से एनएन अनुप्रयोगों से सुसज्जित हैं, जिससे बेहतर उपयोगकर्ता अनुभव और गोपनीयता सुरक्षा के लिए ऊर्जा-कुशल समाधान की आवश्यकता होती है। अध्ययन में एनएन के तीन वर्ग शामिल हैं: सेल्फ ऑर्गनाइजिंग मैप्स (एसओएम), कन्वेन्शनल न्यूरल नेटवर्क्स (सीएनएन), और रिकरंट न्यूरल नेटवर्क्स (आरएनएन), प्रत्येक वर्ग डेटा पुन: उपयोग और ऊर्जा दक्षता के मामले में अद्वितीय चुनौतियां पेश करता है।

थीसिस न्यूरल नेटवर्क मॉडल को पेश करने से शुरू होती है, जिसमें फीडफॉरवर्ड, रिकरंट और डीप न्यूरल नेटवर्क सहित संरचना और विभिन्न प्रकार के एनएन को दर्शाया गया है। एनएन अनुप्रयोगों के विकास और तैनाती चरणों को भी समझाया गया है, जिसमें एज उपकरणों के लिए अनुमान चरण को अनुकूलित करने के महत्व पर जोर दिया गया है।

सीएनएन के संदर्भ में, अनुसंधान विभाजन और शेड्यूलिंग योजनाओं के माध्यम से डेटा के पुन: उपयोग को अनुकूलित करने पर केंद्रित है। सिस्टम-आर्किटेक्चरल बाधाओं और डेटा आकार पर विचार करते हुए, ऑफ-चिप मेमोरी एक्सेस का अनुमान लगाने के लिए एक विश्लेषणात्मक ढांचा विकसित किया गया है। ढांचा विभिन्न विभाजन और शेड्यूलिंग दृष्टिकोणों की तुलना करने में सक्षम बनाता है, जिससे विभिन्न सीएनएन परतों के लिए ऊर्जा दक्षता और थ्रूपुट में सुधार के लिए इष्टतम समाधान की खोज की सुविधा मिलती है।

आरएनएन के लिए एक नया डेटा पुन: उपयोग दृष्टिकोण प्रस्तावित है, जो पिछले समय-चरण गणनाओं पर निर्भरता के कारण अतिरिक्त चुनौतियां पेश करता है। यह कार्य ऑन-चिप मेमोरी क्षमता के बावजूद, किसी भी दो सन्निहित समय चरणों के लिए बड़े मैट्रिक्स के वजन का कुशलतापूर्वक पुन: उपयोग करता है। लॉन्ग-शॉर्ट टर्म मेमोरी नेटवर्क (एलएसटीएम) एक्सेलेरेटर का एफपीजीए कार्यान्वयन लोकप्रिय एलएसटीएम मॉडल के लिए ऊर्जा दक्षता और थ्रूपुट बढ़ाने में इस दृष्टिकोण की प्रभावशीलता को प्रदर्शित करता है।

एसओएम के लिए, अध्ययन सटीकता और ऊर्जा दक्षता पर परिमाणीकरण तकनीकों के प्रभाव का पता लगाता है। एक कस्टम सेमी-सिस्टोलिक सरणी डिज़ाइन तैयार किया गया है, और विभिन्न बिट रिज़ॉल्यूशन कार्यान्वयन के लिए एनएन सटीकता और ऊर्जा खपत के बीच एक व्यापार-बंद का विश्लेषण किया गया है। अध्ययन एसओएम के लिए अलग-अलग बिट रिज़ॉल्यूशन का उपयोग करने के लाभों और व्यापार-बंदों का विश्लेषण करता है, ऊर्जा-सीमित प्रणालियों को पूरा करता है जहां क्षेत्र, शक्ति और प्रदर्शन महत्वपूर्ण विचार हैं।

अंत में, यह पीएच.डी. थीसिस एज उपकरणों के लिए आधुनिक एनएन के ऊर्जा-कुशल त्वरण में अत्याधुनिक को आगे बढ़ाती है। उपन्यास डेटा पुन: उपयोग तकनीकों और विश्लेषणात्मक ढांचे का प्रस्ताव करके, यह शोध अनुमान चरण में सुधार के लिए मूल्यवान अंतर्दृष्टि प्रदान करता है, इस प्रकार ऊर्जा-सीमित वातावरण में एनएन-आधारित अनुप्रयोगों को व्यापक रूप से अपनाने में योगदान देता है। यह कार्य एनएन अनुकूलन और एज एआई अनुप्रयोगों की सीमाओं को आगे बढ़ाते हुए भविष्य के अनुसंधान के लिए नए रास्ते भी खोलता है।

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

The past few years have seen exponential growth in Neural Network (NN) based applications. NN applications are widely used in healthcare, agriculture, road safety, surveillance, defense, number plate identification, medical diagnosis, autonomous driving, recommender systems, and many more. Modern computing systems capable of storing and processing large volumes of data and the availability of big data sets due to digitization have enabled NNs to achieve human-like performance, which was not possible a few decades ago. Their growth was also accelerated by the software libraries like Tensorflow and PyTorch that provide ease of development.

Today, NN-based applications have penetrated our daily lives, e.g., face recognition for authentication, chatbots, shopping recommendations, and photo tagging. Their unprecedented success in solving complicated real-world problems has increased their usage in embedded systems ranging from smartphones and tablets to wearable devices. NNs are also used in the clinical environment, e.g., in bacterial identification and tumor detection. Several of these devices need to be mobile, battery-operated, and should have lightweight. Often, such systems have limited computing resources and tight energy-budget. With time, NNs are growing in size to solve complicated problems and improve accuracy; this trend is expected to continue for several years.

## 1.1 Neural Network Model

Neural Networks are machine learning algorithms inspired by processing mechanisms in the human brain. They can learn from the data using a training process without being programmed explicitly. Inspired by the brain, NNs consist of several neurons connected and organized as layers. There can be several hidden layers in a NN. Figure 1.1 shows a simple NN example.

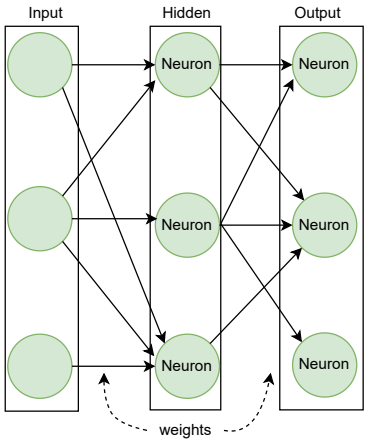



Figure 1.1: Example of simple Neural Network.

Several classes of NNs exist that differ in the number of neurons, connections between them, and training method. Broadly they can be classified as feedforward neural networks (FFNNs) and recurrent neural networks (RNNs) based on the flow on the flow of data between layers. Convolutional Neural Networks (CNNs) and Long short-term memory networks (LSTMs) are quintessential examples of FFNNs and RNNs, respectively. CNNs are mainly used in computer vision-related applications like image classification and object detection, and LSTMs are used in sequential data processing applications like speech recognition and natural language processing (NLP).

On one hand there are single-layer NNs that consist of just two layers - an input and an output layer, with only the output layer performing the computation. Self-Organizing Maps (SOMs) are examples of single-layer NNs and are used in dimensionality reduction and clustering applications. On the other hand, there are modern NN architectures that are very

deep and involve many layers and millions of parameters. Multilayer NNs with three or more hidden layers are often considered as deep neural networks (DNNs). These DNNs are capable of learning complex functions from the data.

## 1.2 Phases of NN Applications: Development to Deployment

NNs need not be programmed explicitly, and they learn from the raw data to provide solutions to real-world problems. For example, NNs first learn to classify an object in an image, in which several example images are provided. This learning process is referred to as training. Once trained, the NN can estimate the output for a new input. Subsequently, the trained NNs are used in applications to estimate the new input's output. This phase is referred to as inference.

Figure 1.2 shows difference phases of NNs from development to deployment. The development phase is usually performed on desktop/laptop machines using deep neural network frameworks, e.g., PyTorch and TensorFlow. After the development phase, NNs

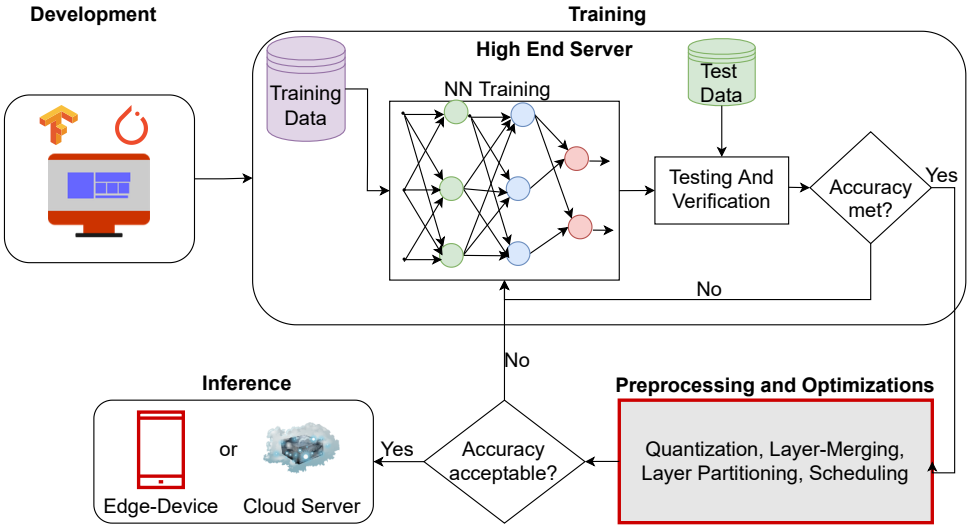



Figure 1.2: Work Flow for Neural Networks.

undergo a training phase where they learn from examples. These DNNs require large training data sets for learning and must go through multiple iterations to achieve the desired accuracy. Training of NNs involves updating weights and biases using large input samples. It is a

repetitive process until the desired accuracy is achieved. Training a DNN may last from a few hours to several weeks. Hence training of DNNs is generally performed on high-performance systems, typically using GPUs.

After the training phase, trained models are used in applications for inferencing, where the deployment platforms may range from cloud to embedded devices. Inference phase computations also involve millions of computations and access large volumes of data. For example, a popular CNN, VGG16, performs 15.47 GMAC operations and accesses 138 million parameters for a single inference. If the NNs are deployed on the cloud, the throughput and energy demands can be met using high-performance systems like GPU. However, deployment on edge devices is challenging due to limited energy and computing resources. Hence optimizing these models before inferencing on edge devices is a must. In this work, we focused on optimizing NN models before deployment on edge devices for inferencing. This phase is shown as *Preprocessing and Optimization* block in Figure 1.2.

## 1.3 Edge AI: Inferencing on Edge Devices

Edge devices are battery-operated with limited resources and a tight energy budget. A few years back, edge devices were primarily used to collect real-world data, and inferences were performed on the cloud. There is a growing trend of shifting the processing of these DNN applications from the cloud to edge devices near the sensors as it improves the user experience, eliminates network bandwidth issues, and improves privacy and security.

Today we see an explosion of applications using deep learning algorithms in consumer electronic devices. Manufacturers are shifting the processing of NNs from cloud to edge devices. Several latest consumer electronic devices, like smartphones, digital cameras, etc., are already equipped with NN applications.

However, DNN inferencing on edge devices is challenging. Figure 1.3 shows the key challenges for Edge AI devices. Energy efficiency and throughput are the two most important

metrics for edge devices. While energy efficiency is paramount for longer battery time, high throughput is desired for better user-response time. Efficient processing of DNNs inferencing on edge devices is critical for their widespread usage.

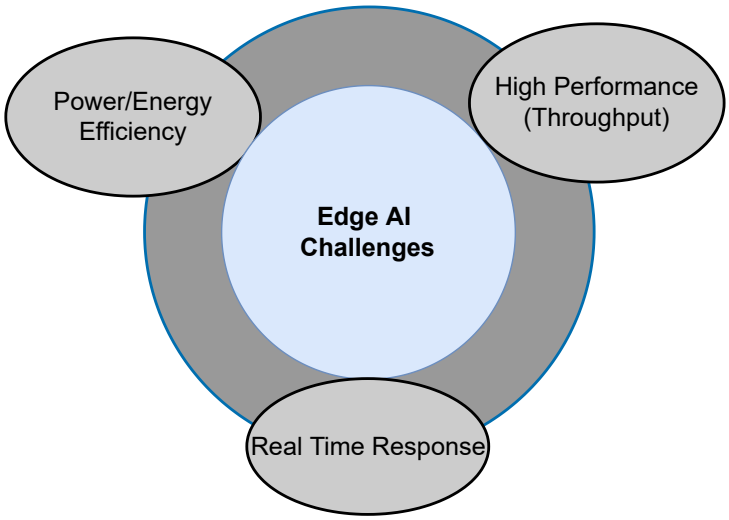



Figure 1.3: Key challenges for Edge-AI Accelerator

Edge devices use customized NN accelerators to meet energy and throughput demands. Figure 1.4 shows a typical DNN accelerator architecture, which consists of an off-chip memory and an accelerator chip. An accelerator chip mainly consists of an on-chip memory of a few hundred KBs and an array of Processing Elements (PEs). The accelerator system has multiple memory levels: off-chip memory, on-chip memory, and the registers inside the PEs. Each memory level has a different storage capacity, access latency, and energy costs. The access energy from off-chip memory is up to two orders of magnitude higher than a PE computation operation [8].

In this thesis, we aimed to improve edge NN accelerators' performance and energy efficiency during the inference phase.

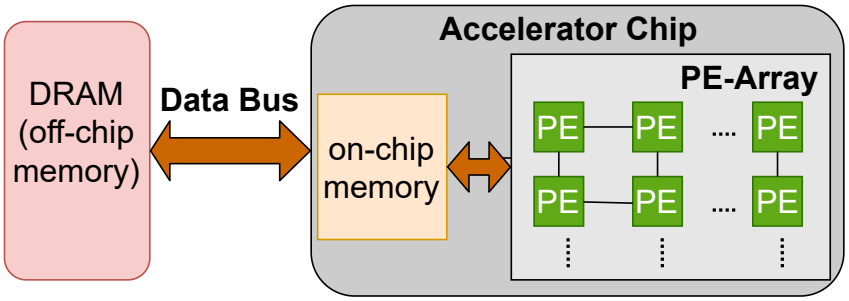



Figure 1.4: Typical DNN accelerator architecture.

## 1.4 Energy Efficient NN Inferencing

### 1.4.1 Previous Work

Several FPGA [2, 16, 18, 50, 51, 55], GPU [10] and ASIC [6, 7, 8, 12] accelerators have been proposed to meet the performance and energy targets. These accelerators apply different techniques to speed up operations. NN computations are memory intensive. With limited on-chip memory on the accelerators and a large difference between latencies and energy consumption of off-chip and on-chip memories, off-chip memory accesses dominate the performance and energy consumption of these accelerators. As much as 80% of the overall energy consumption of an NN accelerator could be due to off-chip memory accesses [6]. Therefore, reducing the off-chip memory is the key to improving the throughput and energy efficiency of DNN accelerators. This has led several researchers to focus on reducing the off-chip memory accesses [6, 9, 55]. Some approaches [14, 27, 42] have used on-chip memory to store all the weights. However, since sizes of weights in modern NN models can be several MBs, these approaches are not scalable and are effective only for small NN models. Approaches attempting to reduce the off-chip memory accesses of NN accelerators can be classified into two broad categories, as shown in Figure 1.5.

One category of work exploits error-tolerance and redundancy in NNs using quantization, compression, and pruning techniques to reduce the precision, the number of operations, and

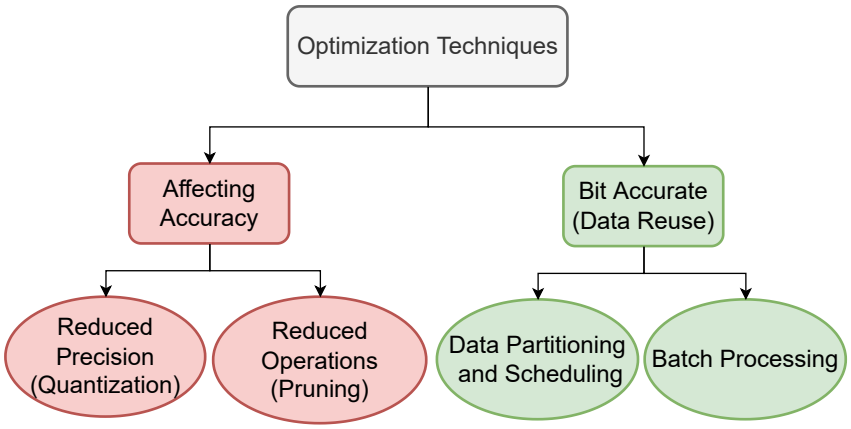



Figure 1.5: Broad Classification of previous works for improving the performance of DNN accelerators.

models' size [5, 14, 20, 27, 48]. With reduced precision, the storage requirement and memory accesses reduce proportionally and improve energy efficiency [46]. Quantization and pruning techniques result in reduced NN model sizes. The reduced model for shallow NNs may fit into the on-chip memory and thus eliminate the off-chip memory bandwidth bottleneck. However, quantization and pruning approaches impact the accuracy of the networks and may only be suitable where accuracy can be compromised. These techniques have been explored only for limited domains such as image processing and computer vision. The number of parameters in modern DNNs is significantly large. For these DNNs, besides quantization and pruning, additional techniques may be required to reduce the off-chip memory accesses further.

The second category of approaches that do not affect the accuracy of the network are data-reuse schemes. The data-reuse schemes are orthogonal to the quantization techniques and can be combined to reduce off-chip memory accesses further. These schemes aim to reduce repeated off-chip accesses to the same NN coefficients when the entire set of NN coefficients does not fit in the on-chip memory. This is quite effective for many modern DNNs (e.g., CNNs, RNNs) with a significantly large number of parameters. One popular scheme which reuses weights is batch processing [28, 41, 55]. During the inference phase, all the inputs use the same weights. In the batch processing scheme, inputs are grouped as a batch and processed

to reuse the weights from the on-chip memory. Increasing the batch size improves the weights reuse but increases the latency, which is undesirable. Secondly, the batch processing scheme only improves the weights reuse, and other layer data types do not get the benefit.

Another well-known data reuse technique is data partitioning and scheduling [28, 55]. In this technique, the data is partitioned into tiles, and operations are scheduled so that data can be accessed from the on-chip memory as far as possible. Modern CNNs have layer data sizes too large to fit into on-chip memory, and it is essential to partition the layer data. There are numerous ways of data partitioning and scheduling for a given CNN layer, offering different extents of data reuse. Choosing an optimal way here is non-trivial.

Loop tiling is applied to partition the layer data, a well-known compiler technique [1]. Conventional problems apply loop tiling where equal data-reuse opportunities exist in all the data dimensions. However, in CNN layers, there are multiple types of data reuse, and the extent of each data reuse varies with different dimensions. Increasing tile dimensions to improve one type of data reuse reduces other types. It is important to consider all the possible types of data reuses of a layer to find the optimal solution. Deciding on the optimal partitioning of NN layers is more challenging than conventional problems.

Compared to FFNNs, RNN computations pose a different challenge for data reuse. RNNs have feedback connections that result in dependency on previous time step computations and limit the possible data reuse. Due to limited on-chip memory of accelerators, the weights are repeatedly accessed from the off-chip memory at each time step. This results in a large volume of off-chip memory access. Que et al. [41] proposed a blocking-batching scheme. However, the benefit of their scheme is limited to matrices involved in time-step independent computations. To handle the dependency problem, Park et al. [37] proposed a time-step interleaved weight reuse scheme (TSI-WR) that reuses the weights in a time-interleaved manner. However, their approach was only partially able to reuse the data, and the extent of reuse depends on the on-chip buffer sizes. Their approach is effective for accelerators with sizeable on-chip memory.

### 1.4.2 Thesis objectives and contributions

The objective of this thesis has been to provide novel energy-efficient solutions for NN accelerators by reducing off-chip memory accesses. The proposed techniques fall within the classification of Figure 1.5 but carry some new ideas. Since every memory access reduction technique is not necessarily applicable to the entire range of NN algorithms, we have considered three NN algorithms that present quite different characteristics from each other. These are Self Organizing Maps (SOMs), Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs).

We have applied quantization techniques on Self Organizing Maps (SOMs) to analyze the impact on NN accuracy and the benefits of improving energy efficiency. These are single layer FFNNs have not been studied in this context previously. We hand-crafted a custom semi-systolic array design for different bit width implementations to analyze the accuracy versus energy trade-off for SOMs.

For multilayer feedforward and recurrent NNs, we have proposed novel data reuse approaches. To find energy-efficient solutions, estimating the off-chip memory accesses is essential. Performing the measurements on the hardware is time-consuming, and vast design space makes it practically impossible. From the description of the NN layer, it is not easy to estimate the off-chip memory accesses. Several factors impact the off-chip memory accesses of the NN layer, which were not considered by the previous works, e.g., bus width and address alignment. In this work, we have developed an analytical framework that computes the off-chip memory accesses of 3D data partitioned into small tiles. The analytical framework considers data shape, tile dimensions, accelerator architecture parameters, and data resolution to compute the off-chip memory accesses.

For CNNs where partitioning and scheduling scheme makes a significant impact, we have analyzed memory accesses of various partitioning and scheduling schemes for memory-intensive layers. There are several possible ways to partition the layer data and

schedule the operations of DNN layers. We expressed determining the optimal solution that minimizes off-chip memory accesses of a NN layer as a constraint optimization problem. We have developed and integrated a model with the analytical framework for computing the off-chip memory accesses of different CNN layers. Previous approaches [28, 55] have ignored the architectural parameters while determining the optimal solution. Our objective in this work is to analyze the impact of architectural parameters on overall off-chip memory access of CNN accelerators and use that information to determine the optimal tile dimensions to reduce the accesses and the total energy of the CNN accelerators.

We implemented the hardware design for memory-intensive CNN layers on FPGA to measure the off-chip memory accesses, latency, run-time, and design power. The implementation is configurable for layer shapes, tile dimensions, on-chip memory sizes, bus width, and data resolution. We measured the energy efficiency and throughput of different approaches using the FPGA design and validated the results of the analytical framework.

Besides CNNs, the other popular class of NN is RNNs. The main challenge in optimizing the off-chip memory accesses of RNNs is the dependency on previous time-step computations. If the accelerator's memory is not large enough to hold weight matrices, this results in a large volume of data accesses from the off-chip memory. To address this, we proposed a novel data reuse approach that reuses all the weights of such matrices for the two consecutive time steps. A characteristic of the proposed approach is that the extent of data reuse is independent of the accelerator's on-chip memory size, making it suitable for LSTM accelerators with small on-chip memory. We implemented the hardware design for LSTMs on FPGA for the proposed, conventional, and state-of-the-art data-reuse approaches. Using the FPGA designs, we estimated the energy efficiency and throughput and demonstrated the efficacy of our approach for popular LSTM models.

## 1.5 Summary and Outline of the Thesis

Previous research has shown that for data partitioning and scheduling, making a choice independently for each layer of a NN is better than making a common choice for the entire NN because of the different shapes of various layers. We observe that the choice of optimal data partitioning and scheduling depends on the shape of a layer and architectural parameters. We present an analytical framework in Chapter 2 that quantifies the off-chip memory accesses for DNN layers of varying shapes, considering the architectural constraints. It helps compare different data partitioning and scheduling schemes to explore the large design space to find the optimal solution for improving the energy and throughput.

Based on the above analytical framework, in Chapter 3, we propose a data reuse approach that considers the architectural parameters and determines the optimal partitioning and scheduling scheme to minimize the off-chip memory access of CNN layers. We demonstrate the efficacy of our partitioning and adaptive scheduling approach on the compute and memory-intensive CNN layers.

Chapter 4 proposes a novel data reuse approach to improve the throughput and energy efficiency of state-of-the-art recurrent neural networks (RNNs). The proposed approach splits the computations and combines them in a way that significantly reduces the off-chip memory accesses of large matrices. We measure the design power and memory accesses on FPGA implementation of Long-Short Term Memory Network (LSTM) accelerators and show our approach's energy and throughput improvements.

We analyze the effect of using different bit resolutions on the accuracy of a NN and the benefits of it for self-organizing maps (SOMs) for designing energy-constrained systems where the area, power, and performance are of critical importance in Chapter 5. Using an efficient implementation of SOM design on FPGA, which can be configured for different bit resolutions, we show performance comparison for different data precisions.

The work done in this thesis improves the state-of-the-art of energy efficient execution of

modern NNs and also gives directions to future research. In Chapter 6, we discuss these new research directions together with the conclusion of our work.

# Chapter 2

# Analyzing Off-chip Memory Accesses

The off-chip memory bandwidth often limits the performance of NN accelerators, and their high energy consumption also results from the large volume of off-chip memory accesses. NN accelerators use on-chip memories and reuse the data from it to minimize the number of off-chip memory accesses. Assuming that the data movement energy primarily depends on the number of bytes accessed from the memory, Figure 2.1 illustrates the impact of data reuse from the on-chip memory on data movement energy. Figure 2.1a and Figure 2.1b show the number of memory accesses and energy estimates when data is directly accessed from the off-chip memory and when reusing the data from on-chip memory, respectively. Let $N$ be the number of bytes accessed by the processing element (PE), and $e_1$ and $e_2$ are the energy per byte access from the off-chip and on-chip memories, respectively. If $E_1$ is the data movement energy when PE accesses $N$ bytes of data directly from off-chip memory (Figure 2.1a), $E_1$ can be expressed as following

$$E_1 = N \cdot e_1 \tag{2.1}$$

Let $E_2$ be the data movement energy when two levels of memory are used ( Figure 2.1b) and $n$ is the data reuse from the on-chip memory per byte of off-chip memory access. In this two-level of memory hierarchy Figure 2.1b, although the total number of bytes accessed by PE is still the

same ($N$), the number of off-chip memory accesses is reduced to $\frac{N}{n}$. The data movement energy ($E_2$ ) in this case can be expressed as follows,

$$\begin{aligned} E_2 &= \frac{N}{n} \cdot e_1 + \frac{N}{n} \cdot n \cdot e_2 \\ &= N \cdot e_1 \cdot (\frac{1}{n} + \frac{e_2}{e_1}) \end{aligned} \tag{2.2}$$

The energy efficiency compared to ( Figure 2.1a) can be expressed as,

$$\begin{aligned} E_{efficiency} &= 1 - \frac{E_2}{E_1} \\ &= 1 - (\frac{1}{n} + \frac{e_2}{e_1}) \end{aligned} \tag{2.3}$$

For a given architecture, $e_1$ and $e_2$ are fixed and $e_1$ is significantly high compared to $e_2$. Equation 2.3 shows energy efficiency improves significantly with data reuse from the lower memory level ($n$). DNN accelerators use multiple levels of on-chip memories and apply techniques to maximize the data reuse from the lower memories to improve energy efficiency.

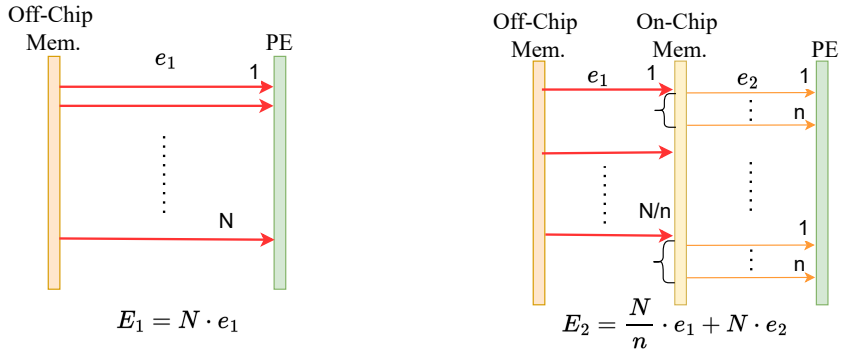



Figure 2.1: Memory accesses and energy estimates for accessing data (a) always from the off-chip memory. (b) from two-level of memory hierarchy while reusing the data from on-chip memory.

The NN accelerator reads the input data (or input activations), filter weights, and partial sums from the off-chip memory and stores them temporarily in the on-chip memory. The PE array reads the data from the on-chip memory to perform the computations and then stores

them back to the on-chip memory. The partial sums or the outputs of the computations are then finally stored in the off-chip memory. The data flow is shown in Figure 2.2. Due to the significant difference between the energy consumption of accessing the data from the off-chip memory and the on-chip memory, NN accelerators aim to minimize the off-chip memory accesses by exploiting the memory hierarchy and data reuse from them.

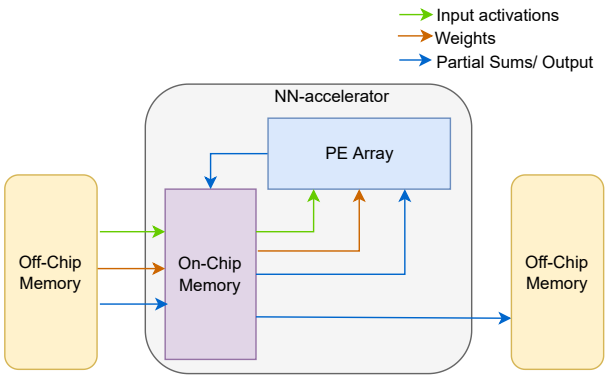



Figure 2.2: Read/Write of inputs, weights, and partial/final outputs from different levels of memories.

The layer data is stored as multi-dimensional arrays in the off-chip memory, which is generally too large to fit in the local on-chip memory. In order to perform the computations, the large layer data is partitioned into small tiles. These tiles are repeatedly fetched from the off-chip memory to compute the final output sum.

To observe the impact of tile dimensions on off-chip memory accesses for a given data reuse scheme, we measured the off-chip memory accesses for a popular CNN, VGG16, for different tile dimensions using the algorithm described in Section 2.2. Figure 2.3 illustrates the impact of tile dimensions on off-chip memory accesses for VGG16. Meanwhile, Figure 2.3a provides a closer look at the significant variations in off-chip memory accesses within a single layer when different tile dimensions are chosen. On the x-axis, we represent a subset of the valid tile search space for the layer, while the y-axis indicates the corresponding off-chip memory access for the selected tile dimensions.

This visualization demonstrates that the choice of tile dimensions plays a crucial role in

determining off-chip memory accesses. A well-selected tile dimension can result in a substantial reduction, up to 90%, in off-chip memory accesses compared to a poorly chosen tile dimension. This observation is further supported by Figure 2.3b, which showcases the computed values for different layers of VGG16. The depicted variations emphasize the importance of optimizing tile dimensions to enhance overall system efficiency and minimize off-chip memory access overhead.

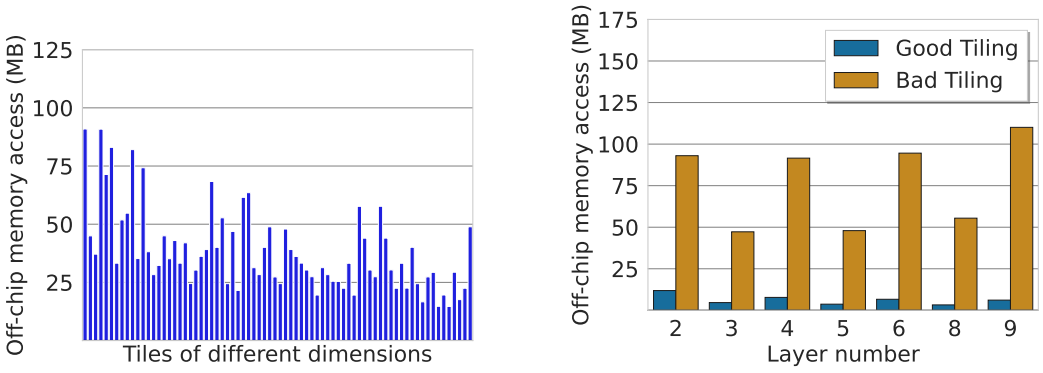



Figure 2.3: Impact of tile dimensions on off-chip memory accesses (a) off-chip memory access variation in a convolution layer when using tiles of different dimensions, x-axis represents the subset of valid tiles of the layer (b) Impact of good and bad tile dimensions on different convolution layers of VGG16.

Determining the optimal tile dimensions and scheduling scheme requires comparing the off-chip memory accesses for different tile dimensions and scheduling schemes. Modern DNNs have a variety of layers (e.g., convolution, fully connected, recurrent, pooling), each exhibiting different types of data access patterns. Even the layers of the same type differ in shape and size. Due to varying layer shapes, sizes, and types, optimal partitioning, and scheduling vary among layers. Finding the optimal partitioning and scheduling scheme by performing the measurements on the hardware is time-consuming, and large search space makes it practically impossible.

To address this, we have developed an analytical framework that integrates models of NN layers to compute a layer's off-chip memory accesses, data access energy, and the number of

compute cycles for mapping a layer on a NN accelerator ( Figure 1.4). Figure 2.4 shows the block diagram of the analytical framework. The framework is used as a design space exploration engine to find the optimal partitioning and scheduling scheme for a given layer type and layer shape to optimize NN accelerators' energy efficiency and throughput.

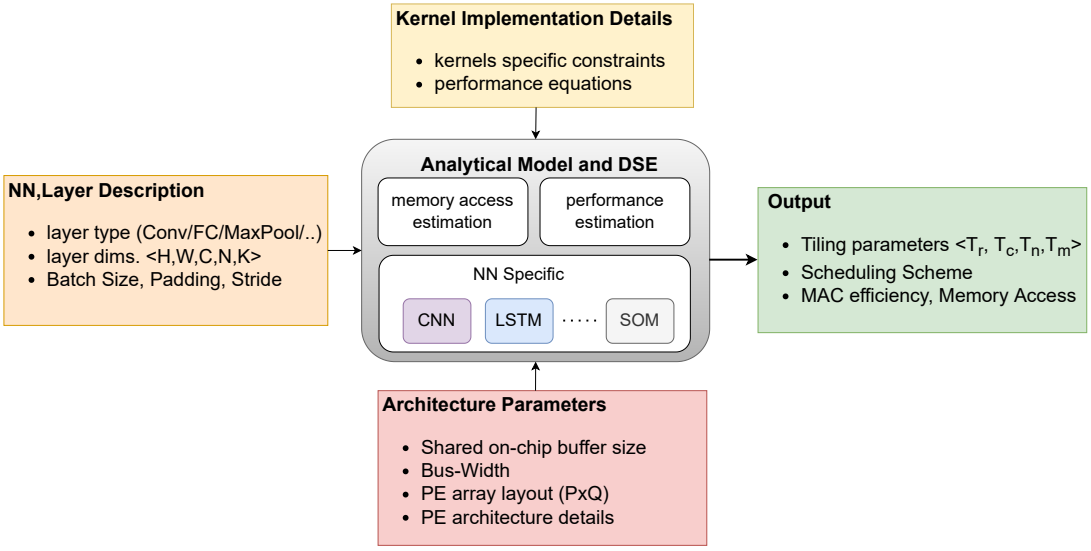



Figure 2.4: Analytical framework to estimate the performance, off-chip memory accesses and energy of DNNs.

If a 3D data is partitioned into $N_{tiles}$ number of tiles, $r_t$ is the trips count and $\mathbb{B}_t$ is the number of bytes accessed from off-chip memory for the $t^{th}$ tile, the off-chip memory access of partitioned 3D data can be computed as follows

$$\mathbb{B}_{3D} = \sum_{t=1}^{N_{tiles}} (\mathbb{B}_t \times r_t) \tag{2.4}$$

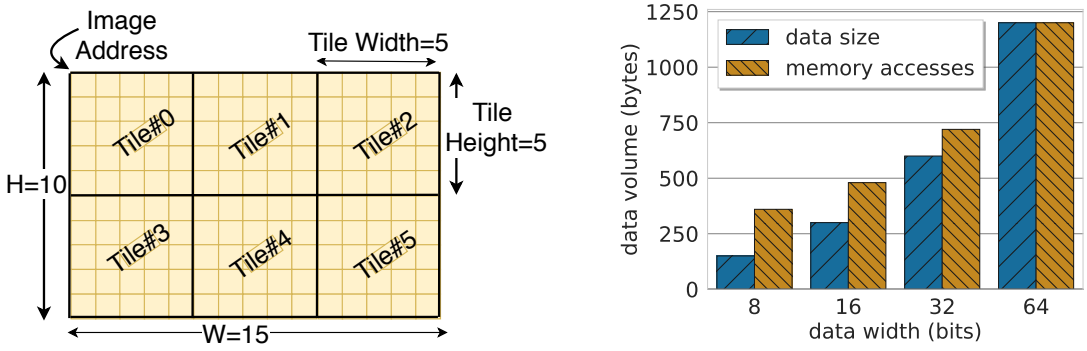
In DNN layers, $r_t$ depends on the layer shape, tile dimensions, and scheduling scheme, and it is the same for all the tiles of the 3D data ($r$). However, $B_t$ in Equation 2.4 varies among the tiles depending on the architectural parameters and address alignment. Equation 2.4 can be

expressed as

$$\mathbb{B}_{3D}{=}r{\times}\sum_{t=1}^{N_{tiles}}\mathbb{B}_t \tag{2.5}$$

The analytical framework computes the off-chip memory accesses of a given 3D data using Equation 2.5, while considering the data resolution and architectural constraints. The framework considers the bus width and data alignment to precisely compute the off-chip memory accesses. It takes the address and shape of the data as input and iterates for all the tile dimensions. The analytical framework integrates models for different layer types and data reuse schemes to compute the memory accesses and data access energy.

## 2.1 Analytical study of architectural parameters




(a) A 2D image and its tiles

(b) Data size vs. memory accesses

Figure 2.5: A 2D data and memory accesses on 64-bit data bus

3D data of DNN layers and the tiles, into which the layer data is partitioned, is composed of stacked 2D frames. Off-chip memory access of the 3D data and its tiles can be computed by adding the off-chip memory accesses of 2D frames. Figure 2.5a shows an example 2D frame of shape 10×15, partitioned into tiles of dimensions 5×5. Although all tiles have the same size, the off-chip memory accesses for different tiles are not the same. To observe the differences between the tile sizes and off-chip memory accesses of the tiles, we implemented a hardware

design using Xilinx SDSoC framework, SDx v2018.3, to access the 2D data stored in DRAM. We measured the memory accesses using AXI Performance Monitor (APM) IP [52], integrated with our design. The target platform is ZedBoard, working at 100MHz frequency, and off-chip memory (DRAM) is accessed using 64 bits AXI bus. Table 2.1 shows each tile's sizes and off-chip memory accesses on 64 bits wide bus for 8 bits data width. The difference between the tile sizes and the memory accesses is due to the bus width and address alignments of different rows of the tile.

Table 2.1: Off-Chip memory accesses of the tiles of size 5×5

| **Tile#** | 0 | 1 | 2 | 3 | 4 | 5 | **Total** |
|---|---|---|---|---|---|---|---|
| **Size(bytes)** | 25 | 25 | 25 | 25 | 25 | 25 | 150 |
| **Mem. Access(bytes)** | 72 | 56 | 56 | 48 | 72 | 56 | 360 |

Figure 2.5b shows the size and off-chip memory accesses of the 2D data shown in Figure 2.5a, for different data bit widths. The ratio of memory accesses to the data size for small data bit width is high. Modern NN accelerators use a wide memory bus to improve the memory bandwidth and a low number of bits to represent the data to reduce the storage requirements and memory accesses. Therefore it is crucial to consider the architectural parameters for low-resolution data.

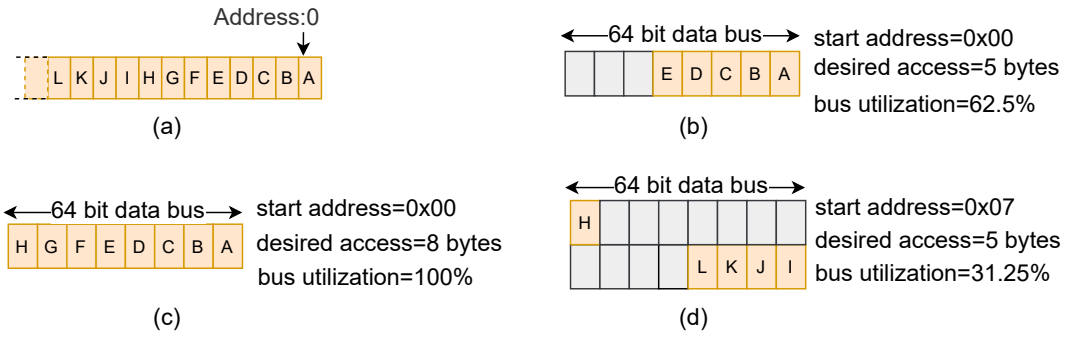



Figure 2.6: Off-chip memory accesses on 64-bit wide data bus

The DNN accelerators use a wide data bus to access off-chip memory to meet the high memory bandwidth requirement [6, 8]. If the number of bytes accessed from an off-chip memory address is not a multiple of bus width or the address is not aligned to the word

boundary, it results in unused bytes lanes of the data bus. Figure 2.6 illustrates memory accesses on a 64-bit data bus. Figure 2.6a shows a read transaction of 8 bytes from an aligned address and uses the full bus width. However, if only 5 bytes are read from an aligned address, as shown in Figure 2.6b, 8 bytes are still accessed. If 5 bytes are read from an unaligned address, it results in 16 bytes of data access, as shown in Figure 2.6c. The unused byte lanes do not carry any useful data, but contribute to overall energy consumption. The length of the data read should be chosen such that bus utilization is high and off-chip memory accesses and energy consumption are minimized.

### 2.1.1 Aligned Access

Number of bytes accessed from off-chip memory ($\mathbb{B}_{align}$) from aligned addresses for different data length ($l$) is in multiples of bus width ($BW$), e.g., reading 10 bytes results in 16 bytes of off-chip memory access. $\mathbb{B}_{align}$ for $l$ bytes and bus width $BW$ can be expressed as

$$\mathbb{B}_{align}(l, BW) = \lceil \frac{l}{BW} \rceil \times BW \tag{2.6}$$

### 2.1.2 General Case

To analyze the effect of bus width and address alignment, we express the number of bytes accessed from off-chip memory ($\mathbb{B}$) for $l$ bytes of data as a function of $l$, its off-chip memory address ($Addr$), and the bus width ($BW$). The off-chip memory bus protocol supports burst based transactions where multiple data transfers of *transfer size* happen from a starting address [3]. Most transfers in a transaction are aligned to the *transfer size*. However, the first transfer may be unaligned to the word boundary.

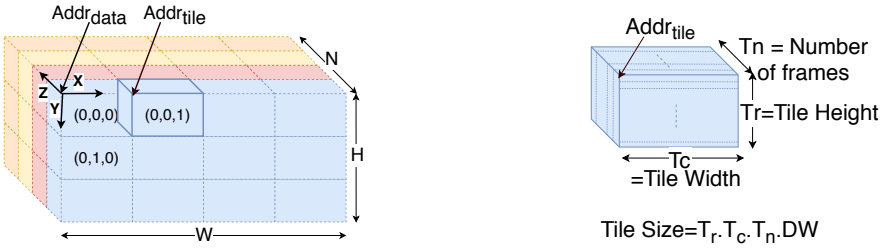
The number of bytes accessed from off-chip memory for accessing $l$ bytes from address

$Addr$ on $BW$ bytes of bus width can be expressed as

$$\mathbb{B}(Addr, l, BW) = (\lceil \frac{Addr + l}{BW} \rceil - \lfloor \frac{Addr}{BW} \rfloor) \cdot BW \tag{2.7}$$

Equation 2.6 is a special case of Equation 2.7 when memory access is from an aligned address.

## 2.2 Off-Chip Memory Access of 3D Data




(a) A 3D data partitioned into tiles (b) A 3D tile

Figure 2.7: Off-Chip Accesses and a 3D partitioned data and tiles

Consider a 3D data of shape $\langle W, H, N \rangle$, partitioned into tiles of dimension $\langle T_c, T_r, T_n \rangle$ as shown in Figure 2.7a and Figure 2.7b. Each tile is identified by index $(x, y, z)$ where $x, y, z \in \mathbb{Z}$ and

$$\begin{aligned} 0 \leq & x < \lceil \frac{W}{T_c - \delta} \rceil, \\ 0 \leq & y < \lceil \frac{H}{T_r - \delta} \rceil, \\ 0 \leq & z < \lceil \frac{N}{T_n} \rceil \end{aligned} \tag{2.8}$$

$\delta$ represents the number of elements or rows that overlap between adjacent tiles. In the context of Convolutional Neural Networks (CNNs) when accessing input feature maps ($ifm$) with a stride ($S$) less than the filter size ($K$), there occurs an overlap between adjacent tiles. The computation of $\delta$ is given by $\delta = (K - S)$, where $K$ denotes the filter dimensions and $S$ is the filter stride. This overlap introduces a reuse opportunity known as Inter-tile overlap reuse. In

our implementation, each tile is fetched separately, and we have not accounted for previously fetched data. This simplification is deliberate to avoid implementation complexity, with the assumption that any negligible impact on overall memory access can be safely disregarded.

Algorithm 1 computes the number of bytes accessed from off-chip memory for the 3D data. Each data element is represented by $DW$ bytes. The algorithm takes the address of the data ($Addr_{data}$), tile dimensions, data shape, and $\delta$ as input and iterates lines 4–10 for all the tiles. It computes indices of the first element of the tile in the 3D array of the data using the tile index $(x, y, z)$ and $\delta$ in lines 5–7. Using the indices of the first element, it computes the address of the tile in line 8 and the dimensions of the tile in line 9. The algorithm computes the total number of bytes accessed from off-chip memory by accumulating the number of bytes accessed for each tile in line 10.

**Algorithm 1** BW Aware off-chip memory access

**Inputs:**
$Addr_{data}$:Starting address of the data in off-chip memory
$\langle T_c, T_r, T_n \rangle$: Tile dimensions
$\langle W, H, N \rangle$: Input data shape
$\delta$: Number of overlapped elements between consecutive tiles
**Output:**
$Acc_{data}$: Number of bytes accessed from off-chip memory

1: **procedure** BWA($Addr_{data}, \langle T_c, T_r, T_n \rangle, \langle W, H, N \rangle, \delta$)
2: $Acc_{data} \leftarrow 0$
3: $Addr_{tile} \leftarrow Addr_{data}$
4: **for** $x, y, z \leftarrow (0,0,0)$ **to** $(\lceil \frac{W}{T_c - \delta} \rceil - 1, \lceil \frac{H}{T_r - \delta} \rceil - 1, \lceil \frac{N}{T_n} \rceil - 1)$ **do**
5: $c \leftarrow x \cdot (T_c - \delta)$
6: $r \leftarrow y \cdot (T_r - \delta)$
7: $n \leftarrow z \cdot T_n$
8: $Addr_{tile} \leftarrow$ GETTILEADDR($Addr_{data}, c, r, n, \langle W, H, N \rangle$)
9: $\langle T_c^1, T_r^1, T_n^1 \rangle \leftarrow$ GETTILEDIM($c, r, n, \langle T_c, T_r, T_n \rangle, \langle W, H, N \rangle$)
10: $Acc_{data} \leftarrow Acc_{data} +$ GETTILEACC($Addr_{tile}, \langle T_c^1, T_r^1, T_n^1 \rangle, \langle W, H, N \rangle$)
11: **return** $Acc_{data}$
12: **procedure** GETTILEACC(Addr$_{tile}$,$\langle T_c, T_r, T_n \rangle$,$\langle W, H, N \rangle$)
13: $Acc_{tile} \leftarrow 0$
14: **if** $T_c = W$ **and** $T_r = H$ **then**
15: $contSz \leftarrow T_c \cdot T_r \cdot T_n \cdot DW$
16: $Acc_{tile} \leftarrow \mathbb{B}(Addr_{tile}, contSz, BW)$
17: **else**
18: **for** $f \leftarrow 1$ **to** $T_n$ **do**
19: Addr$_f \leftarrow Addr_{tile} + (f-1) \cdot H \cdot W$
20: **if** $T_c = W$ **then**
21: $contSz \leftarrow T_c \cdot T_r \cdot DW$
22: $Acc_{tile} \leftarrow Acc_{tile} + \mathbb{B}(Addr_f, contSz, BW)$
23: **else**
24: $contSz \leftarrow T_c \cdot DW$
25: **for** $r \leftarrow 1$ **to** $T_r$ **do**
26: $Addr_{(f,r)} \leftarrow Addr_f + (r-1) \cdot W$
27: $Acc_{tile} \leftarrow Acc_{tile} + \mathbb{B}(Addr_{(f,r)}, contSz, BW)$
28: **return** $Acc_{tile}$
29: **procedure** GETTILEDIM($c, r, n$,$\langle T_c, T_r, T_n \rangle$,$\langle W, H, N \rangle$)
30: $T_c^1 \leftarrow min(T_c, W - c)$
31: $T_r^1 \leftarrow min(T_r, H - r)$
32: $T_n^1 \leftarrow min(T_n, N - n)$
33: **return** $\langle T_c^1, T_r^1, T_n^1 \rangle$

The address of the tile in line 8 is computed by the function GetTileAddr using Equation 2.9 below

$$GetTileAddr(Addr_{data}, c, r, n, \langle W, H, N\rangle) = Addr_{data} + c + r \cdot W + n \cdot W \cdot H \tag{2.9}$$

If the data dimensions are not multiples of corresponding tile dimensions, the tiles at the border of the data (e.g., the right-most or bottom-most tiles) will have smaller dimensions. The function GetTileDim computes the dimensions of the tiles.

The function GetTileAcc in Lines 12–27 computes the addresses and number of bytes ($contSz$) for each off-chip memory transaction. $contSz$ is the number of contiguous bytes to access in a transaction. Let $T_n$ be the number of frames in the tile. If the tile width ($T_c$) is the same as the data width ($W$) (line 20), then all the data elements of $T_r$ rows of a frame are contiguous, and the complete tile can be accessed using $T_n$ number of addresses. If the condition in line 14 is true, then all elements of the tile are contiguous and can be accessed using a single transaction of $T_c \cdot T_r \cdot T_n \cdot DW$ bytes (line 15). In all other cases, the tile is accessed using row addresses computed in line 26. The function computes the number of bytes accessed from off-chip memory for a tile ($Acc_{tile}$) in lines 16, 22, and 27 using Equation 2.7.

DNN accelerators access layer data (ifm, ofm, and weights) partitioned into tiles. The tile dimensions should be chosen carefully to minimize the transfer of extra bytes to reduce off-chip memory access. We have proposed a bus width aware approach (BWA) that factors in the architectural parameters to precisely compute the off-chip memory access of 3D tiles and the layer data.

## 2.3 Implementation and Results

We've implemented a tool that calculates off-chip memory access for 3D data using the BWA algorithm ( Algorithm 1). This tool, which considers tile dimensions, layer shape, and the count

of overlapping elements as inputs, is adaptable for different bus widths and data bit widths. Executed on a desktop machine, the tool quickly analyzes off-chip memory accesses for various tile dimensions. It seamlessly integrates with our analytical framework, estimating off-chip memory accesses, latencies and data movement energies of different NN layers across.

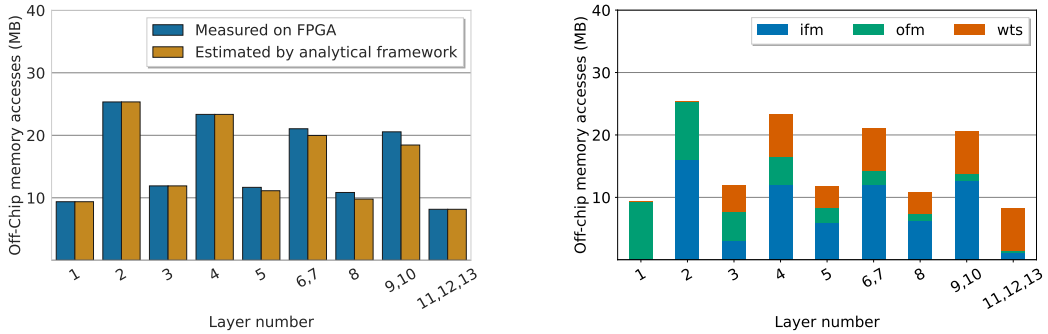
Validation of the off-chip memory access calculations of the tool was conducted by accessing the same shape of 3D data and using the same tile dimensions on a Xilinx FPGA. FPGA implementation was done using the Xilinx SDSoC framework, SDx v2018.3. The FPGA implementation utilized SDx pragmas for zero_copy data movement and integrated the AXI Performance Monitor (APM) IP [52] for real-time performance metrics, including bus latency and memory traffic. Executed on the ZedBoard platform at a 100MHz frequency, the off-chip memory (DRAM) was accessed using a 64-bit AXI bus. The FPGA implementation, taking 3D shape and tile dimensions as inputs, accessed 3D data from DRAM using loop tiling. Validation was performed by comparing the tool's calculated off-chip memory bytes with the APM-logged values for various 3D data shapes and tile dimensions.

### 2.3.1 Results

Figure 2.8a illustrates the comparison between off-chip memory accesses computed by the analytical framework and measurements obtained on a Xilinx FPGA. For various layers of different shapes and sizes in the popular CNN, VGG16, reveal that the variation between off-chip memory accesses calculated by the analytical framework and the measured values on FPGA is consistently less than 4%.

Within the Convolution layers, three types of data are involved: filter weights ($wts$), input feature maps ($ifm$), and output feature maps ($ofm$). Our framework proves instrumental in not only accurately computing the off-chip memory accesses but also in analyzing the layer-wise distribution and breakdown of memory accesses for different types of data, filter weights ($wts$), input feature maps ($ifm$), and output feature maps ($ofm$), as depicted in Figure 2.8b.

The proposed framework is a valuable tool for quickly analyzing the memory accesses and data access energy for different tile dimensions and data reuse schemes to search for the optimal solution.




(a) (b)

Figure 2.8: Off-chip memory accesss of VGG16 layers (a) Comparison between the off-chip memory accesses estimated by the analytical framework and measured on FPGA. (b) Breakdown of off-chip memory access by data type estimated by the analytical framework.

## 2.4 Summary

The chapter investigates the critical role of off-chip memory access in neural network (NN) accelerators, highlighting its impact on energy efficiency. The chapter addresses challenges arising from limited off-chip memory bandwidth and the energy-intensive nature of these accesses in NN accelerators. It introduces a sophisticated analytical framework that combines NN layer models to estimate performance metrics, off-chip memory accesses, and energy consumption. The proposed Bus Width Aware algorithm offers a precise method for computing off-chip memory accesses in 3D data, considering tile dimensions, layer shapes, and bus widths. The analytical tool's accuracy is validated through FPGA measurements, demonstrating less than 4% variation. Overall, the chapter emphasizes the significance of optimizing memory access efficiency for NN accelerators and contributes a robust framework for achieving this goal.

# Chapter 3

# Optimizing the Performance of CNN Accelerators

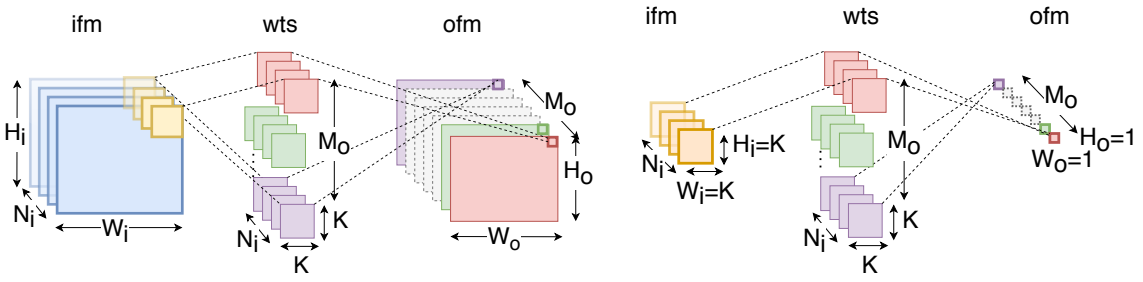
CNNs are the state of the art machine learning algorithms. CNNs can achieve human-like accuracy in computer vision-related tasks. In order to achieve high accuracy, modern CNNs use a deep hierarchy of layers and perform compute-intensive and memory-intensive operations. CNN accelerators use many processing elements to exploit parallelism to speed up the computations. However, limited off-chip memory bandwidth limits their performance. In addition, considerable data transfer volume from the off-chip memory also results in high energy consumption.

CNNs have a sequence of mainly three types of layers: convolution layer (CL), pooling layer, and fully connected layer (FCL). There are several CLs, and a pooling layer usually follows each CL. The last few layers of the CNNs are FCLs. VGG16 has thirteen CLs, and the last three layers are FC. Similarly, AlexNet has five CLs, followed by 3 FCLs. Pooling layer computations involve sliding a two-dimensional filter window over a single channel of *ifm* and selecting one activation from the window using an operation like maximum or average. There are no parameters in pooling layers. The pooling layer helps reduce the activation sizes and, thus, the number of parameters in subsequent layers. CL and FC layers are compute-intensive,

requiring millions of parameters.

Figure 3.1a and Figure 3.1b illustrate the computations of a CL and an FCL, respectively. Each CL and FCL layer takes 3D input frames (*ifm*) and applies filter weights (*wts*) to compute output frames (*ofm*). 3D $ifm$, $wts$ and $ofm$ are composed of $2D$ matrices stacked together in depth dimension. The number of 2D matrices is referred to as channels or depth of the data type. The number of channels ($N_i$) in each filter and $ifm$ are the same.

The convolution is performed by element-wise product between $K \times K$ filter elements with a 2D window of $ifm$ of size $K \times K$, for all the $N_i$ channels and then adding resulting $K \times K \times N_i$ elements to compute one element of the $ofm$ channel. Other elements of the $ofm$ channel are computed by sliding the 2D filter over the spatial dimension of the $ifm$ ($H_i \times W_i$). Filter slides by S elements from left to right and then top to bottom. $K$ and $S$ are referred to as the filter size and stride of the filter, respectively. We assumed square filters of dimension $K \times K$, $K$ is an odd integer, filter stride in horizontal and vertical directions are identical, and $S \leq K$, which are typically the case.




(a) Convolution Layer (b) Fully Connected Layer

Figure 3.1: Convolution and fully connected layers

$ofm$ dimensions of CL and FC layers can be computed using the $ifm$ dimensions, filter size ($K$), and stride ($S$). Figure 3.2 illustrates the relation of ofm dimension with ifm and filter dimensions for stride $S$=1. The filter can be placed at $W_i - (K-1)$ horizontal locations and $H_i - (K-1)$ locations in the vertical dimensions of the $ifm$. If $S$=1, the $ofm$ spatial dimensions are $K-1$ elements less than the $ifm$ dimensions. Several CNN architectures add padding elements to the border of the $ifm$ spatial dimensions. If the padding on the left, right,

top, and bottom are $P_l$, $P_r$, $P_t$, and $P_b$, respectively, the $ofm$ dimensions for $S$=1 can be computed as follows,

$$W_o = W_i + P_l + P_r - (K-1)$$
$$H_o = H_i + P_t + P_b - (K-1) \tag{3.1}$$

If the stride is not 1, the number of elements in the $ofm$ spatial dimensions reduces by a factor

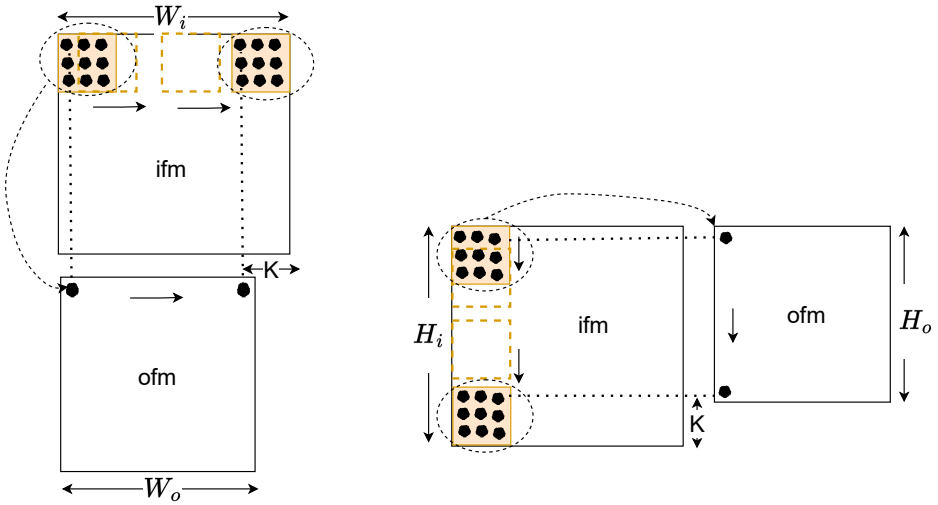



Figure 3.2: Input and Output dimensions of CL layer for filter stride=1

of $S$. If the filter stride is $S$, $ofm$ spatial dimensions can be expressed as follows, which is the generalization of the equation Equation 3.1

$$W_o = \lceil \frac{W_i + P_l + P_r - (K-1)}{S} \rceil$$
$$H_o = \lceil \frac{H_i + P_t + P_b - (K-1)}{S} \rceil \tag{3.2}$$

If the padding at the top, bottom, left and right are the same as $P$, Equation 3.2 can be simplified

as follows,

$$W_o = \frac{W_i + 2 \cdot P - K}{S} + 1$$
$$H_o = \frac{H_i + 2 \cdot P - K}{S} + 1 \tag{3.3}$$

CNN accelerators have limited on-chip memory size. Layer activations and parameter sizes are too large to fit into the on-chip memory. CNN accelerators apply loop tiling to partition the layer data into small tiles that fit into on-chip memory. Loop tiling is a compiler technique [1] that partitions the loop iteration space and large arrays into smaller tiles to increase the data locality and ensure data fits into smaller memories. Figure 3.3 shows a layer's data stored in off-chip memory and its tiles in the accelerator's on-chip buffer.

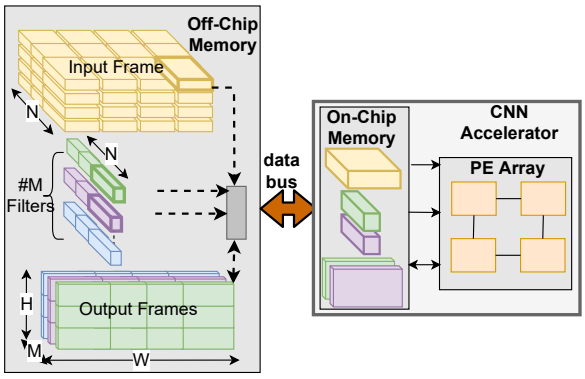



Figure 3.3: CNN layer tiles in off-chip and on-chip memory

Listing 3.1 shows the Pseudo code of the tiled version of a convolution layer. The computations involve nested loops. The outer five loops labeled as $L_D, L_H, L_W, L_M, L_N$ select the tiles of the data, and inner loops perform computations on the selected tiles. The order of the outer loops decides the sequence in which different tiles are processed. Each of the $5!$ permutations of the outer loop results in a valid schedule.

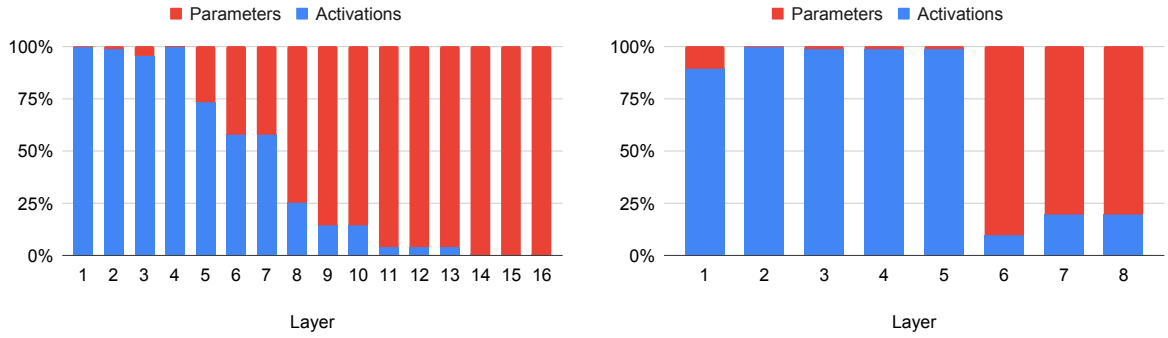
There are $4!$ permutations in which loop $L_M$ appears as the innermost loop. In these permutations, all iterations of the innermost loop $L_M$ use the same *ifm* data. In all such loop orderings, the load statement of *ifm* tile can be moved before the innermost loop $L_M$ and

```
L_D: for(d=0;d<D;d++) {
  L_H: for(row=0;row<H;row+=Tr) {
    L_W: for(col=0;col<W;col+=Tc) {
      L_N: for(ti=0;ti<N;ti+=Tn) {
        L_M: for(to=0;to<M;to+=Tm) {
          //load wts tile
          //load ifm tile
          //load ofm tile
          for(trr=row;trr<min(row+Tr,H);trr++) {
            for(tcc=col;tcc<min(col+Tc,W);tcc++) {
              for(too=to;too<min(to+Tm,M);too++) {
                for(tii=ti;tii<min(ti+Tn,N);tii++) {
                  for(i=0;i<K;i++) {
                    for(j=0;j<K;j++) {
                      ofm[d][to][row][col] += weights[to][ti][i][j] *
                                              ifm[d][ti][S*row+i][S*col+j];
          } } } } } }
          //store ofm tile}
} } } }}
```

Listing 3.1: Pseudo code of a tiled convolution layer

reused in all its iterations [55]. The movement of ifm tile loading reduces the number of off-chip accesses of ifm tile by a factor of $\frac{M}{T_m}$. This ordering scheme exploits ifm data reuse and is referred to as Input reuse-oriented scheme (IRO). Similarly, the set of $4!$ permutations in which loop $L_N$ is an innermost loop exploits the data reuse of *ofm* and is referred to as output reuse oriented (ORO). The remaining $3 \times 4!$ permutations in which loops $L_D, L_R, L_C$ appear as the inner loops exploit the weight data reuse and are referred to as the Weight Reuse Oriented scheme(WRO). The amount of data reused in different schemes varies with layer shape.

CNN's layers have varying shapes. Figure 3.4 shows the parameters and activation proportions in convolution (CL) and fully connected layers (FC) of VGG16 and AlexNet. The first few layers have a large volume of activations (*ifm* and *ofm*), and the last few CLs and FCLs have a large volume of parameters(*wts* and *biases*). Scheduling schemes that optimize the off-chip memory access of activations will work well in the first few layers but may not work well for deeper layers with a large volume of *wts*. The scheduling scheme optimal for one layer may be suboptimal for other layers. Therefore, each layer needs to be analyzed here.




(a) VGG16 (b) AlexNet

Figure 3.4: Parameters and activations proportions in CL and FC layers of CNNs.

## 3.1 Related Work

Several studies ([28, 30, 34, 38, 39, 43, 55]) have addressed the substantial impact of data transfer on the performance and energy efficiency of DNN accelerators. These works aimed to optimize off-chip memory accesses through techniques such as loop ordering and loop tiling.

Peeman et al. [38] presented an analytical method to optimize the convolution nested loops for inter-tile data reuse using loop transformations. Inter-tile reuse scheme proposed by them creates the dependencies between the tiles and reduces the inter-tile parallelism. Ma et al. [30], opted to ignore tile overlap, utilizing a fixed loop ordering and loop tiling scheme. They stored the full depth of $ifm$ tiles to minimize partial sum access from off-chip memory, but this approach may not scale for layers with numerous feature maps.

Shi et al. [43] proposed intra-output feature map parallelism to leverage data locality, aiming to reduce off-chip DRAM accesses by reusing data from on-chip buffers. However, their design has limitations, including the use of separate buffers for input and output feature maps, reducing flexibility. Additionally, their analysis focused solely on height and width tile dimensions, employing a fixed loop ordering and tiling scheme that may not be optimal for various layer shapes in CNNs.

Zhang et al. [55] expressed off-chip memory access as a function of tile dimensions and layer shape, determining optimal tile dimensions by enumerating legal options. They sought

a global optimal tile dimension to simplify hardware design, utilizing a common data reuse scheme for all layers. However, due to varying layer shapes, the optimal tile dimensions and data-reuse schemes differ across layers.

Motamedi et al. [34] considered the impact of adaptive tiling and demonstrated its superior performance compared to static solutions. However, their adaptive tiling was limited to height and width dimensions of feature maps.

The closest approach to ours is the layer-wise adaptive data partitioning and scheduling scheme by Li et al. [28]. Nevertheless, their method overlooked architectural parameters and address alignment, assuming identical off-chip memory accesses for tiles of the same dimensions. This assumption may lead to suboptimal tile dimensions in their approach.

Our approach considers an adaptive strategy for loop tiling and loop orderding for all the layers, analyzing the impact of architectural parameters on off-chip memory access. We use this information to determine optimal tile dimensions and data reuse scheme, aiming to reduce off-chip memory accesses for different layers of CNNs.

## 3.2 Off-Chip Memory Accesses of CNN Layers

CNN accelerators access 3D data of *ifm, ofm*, and *wts* of each layer partitioned into tiles. It accesses tiles of the layer from off-chip memory one or multiple times. Trip counts of the tiles depend on the layer shape, tile dimensions, and the data reuse scheme. If the batch size is $D$, layer shape is $\langle W_o, H_o, N_i, M_o \rangle$, and tiling parameters are $\langle T_{c_o}, T_{r_o}, T_{n_i}, T_{m_o} \rangle$, trip counts of tiles in IRO, ORO, and WRO schemes can be determined using layer shape and tiling parameters.

### 3.2.1 Trip Counts of Tiles

Figure 3.5 shows a toy convolution layer data partitioned into tiles. The $ifm$ is partitioned into two tiles in spatial dimension and two in depth. There are two filters $wts_C$ and $wts_D$ in

the layer, and each is partitioned into two tiles in depth. The convolution output is computed and stored as 4 $ofm$ tiles. Figure 3.6 shows three different valid schedules for the tiles of the

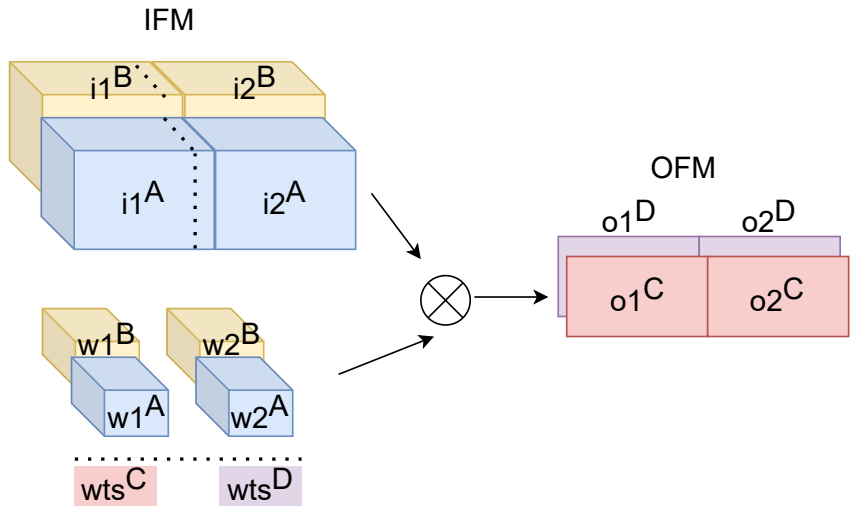
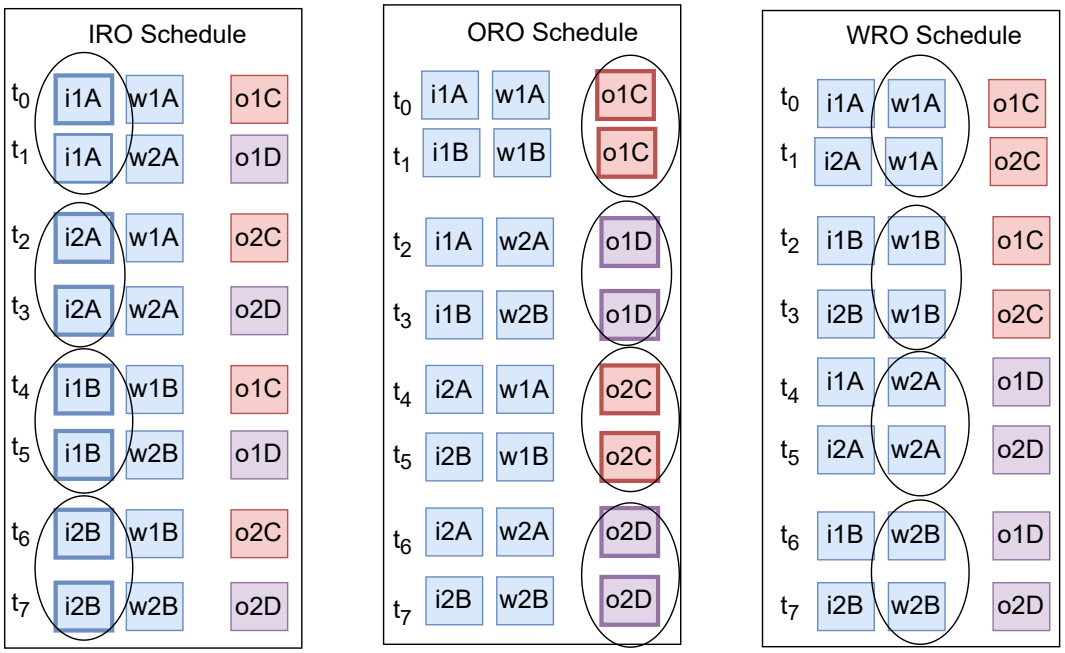



Figure 3.5: A Convolution layer data partitioned into tiles

convolution layer shown in Figure 3.5.




(a) IRO Scheduling (b) ORO Scheduling (c) WRO Scheduling

Figure 3.6: Different Scheduling of tiles.

There are several ways in which a combination of $ifm$, $ofm$, and $wts$ tiles can be accessed

from the off-chip memory to perform the computations by the accelerator, resulting in a valid schedule. These schedules differ in how tile combinations are fetched from the off-chip memory and the volume of off-chip memory accesses. Figure 3.6 shows three different schedules, which minimizes the off-chip memory accesses of one data type.

Figure 3.6a shows the Input Reuse Oriented (IRO) scheme schedule, which maximizes the input data reuse. In the IRO scheduling scheme, all the operations involving a given $ifm$ tile are scheduled consecutively to access each $ifm$ tile only once from the off-chip memory. There are $\lceil \frac{H_o}{T_{r_o}} \rceil \lceil \frac{W_o}{T_{c_o}} \rceil$ number of $ifm$ tiles in spatial dimension and $\lceil \frac{N_i}{T_{n_i}} \rceil$ number of $ifm$ tiles in depth dimension. Each $wts$ tile is accessed repeatedly from the off-chip memory to convolve with different spatial $ifm$ tiles, and each $ofm$ tile is accessed repeatedly to compute the final outputs using the $ifm$ tiles in the depth dimension. Unlike the $wts$ tile, which are only read from the off-chip memory, each $ofm$ tile needs to be written to and read from the off-chip memory, except the first time when it is only written. The trip count of $ifm$, $ofm$ and $wts$ tiles in IRO scheme can be computed as following,

$$\begin{aligned} r_{ifm} &= D \\ r_{ofm} &= (2\lceil \frac{N_i}{T_{n_i}} \rceil - 1)D \\ r_{wts} &= \lceil \frac{H_o}{T_{r_o}} \rceil \lceil \frac{W_o}{T_{c_o}} \rceil D \end{aligned} \tag{3.4}$$

Figure 3.6b shows the Output Reuse Oriented (ORO) scheme, which maximizes the reuse of partial sums. In the ORO scheduling scheme, all the partial sums required for an $ofm$ tile are performed consecutively, and the final sum is written once to the off-chip memory. Computing a single $ofm$ tile requires convolving all the $ifm$ and $wts$ tiles in the input depth dimension. This results in repeatedly accessing the $ifm$ and $wts$ tiles from the off-chip memory. In this scheme, each $ifm$ tile is accessed $\lceil \frac{M_o}{T_{m_o}} \rceil$ times to convolve with all the filters and each $wts$ tile is accessed $\lceil \frac{H_o}{T_{r_o}} \rceil \lceil \frac{W_o}{T_{c_o}} \rceil$ times to perform the operations with all the $ifm$ tiles in spatial dimensions. The trip count of $ifm$, $ofm$, and $wts$ tiles for the ORO scheduling scheme can be

computed as following,

$$\begin{aligned} r_{ifm} &= \lceil \frac{M_o}{T_{m_o}} \rceil D \\ r_{ofm} &= D \\ r_{wts} &= \lceil \frac{H_o}{T_{r_o}} \rceil \lceil \frac{W_o}{T_{c_o}} \rceil D \end{aligned} \tag{3.5}$$

Figure 3.6c shows the Weight Reuse Oriented (WRO) scheme that maximizes the weights reuse. Each $wts$ tile is accessed once from the off-chip memory, and all the operations involving the $wts$ tile are performed consecutively. This requires repeatedly accessing the $ifm$ and $ofm$ tiles from the off-chip memory. Each $ifm$ tile is accessed $\lceil \frac{M_o}{T_{m_o}} \rceil$ times, once for each $wts$ tile. There are $\lceil \frac{N_i}{T_{n_i}} \rceil$ tiles in the input depth. The partial sums resulting from each $ifm$ tile in the input depth need to be read from and written to off-chip memory, except for the first tile, which is only written to the off-chip memory. Total trip counts of the output partial sums tiles are $(2\lceil \frac{N_i}{T_{n_i}} \rceil - 1)$. In this case, the trip count of $ifm$, $ofm$ and $wts$ tiles can be computed as following,

$$\begin{aligned} r_{ifm} &= \lceil \frac{M_o}{T_{m_o}} \rceil D \\ r_{ofm} &= (2\lceil \frac{N_i}{T_{n_i}} \rceil - 1) D \\ r_{wts} &= 1 \end{aligned} \tag{3.6}$$

Trip counts of $ifm$, $ofm$, and $wts$ tiles in IRO, ORO, and WRO schemes can be expressed as the rows of the matrix $\mathbf{R}$ using Equation 3.4, Equation 3.5, and Equation 3.6, where columns

represent *ifm, ofm*, and *wts* trip counts as shown in Equation 3.7 below.

$$\mathbf{R} = \begin{bmatrix} \mathbf{r}_{iro} \\ \mathbf{r}_{oro} \\ \mathbf{r}_{wro} \end{bmatrix} = \begin{bmatrix} D & (2\lceil \frac{N_i}{T_{n_i}} \rceil - 1)D & \lceil \frac{H_o}{T_{r_o}} \rceil \lceil \frac{W_o}{T_{c_o}} \rceil D \\ \lceil \frac{M_o}{T_{m_o}} \rceil D & D & \lceil \frac{H_o}{T_{r_o}} \rceil \lceil \frac{W_o}{T_{c_o}} \rceil D \\ \lceil \frac{M_o}{T_{m_o}} \rceil D & (2\lceil \frac{N_i}{T_{n_i}} \rceil - 1)D & 1 \end{bmatrix} \tag{3.7}$$

### 3.2.2 Off-chip memory access of CL

To compute the off-chip memory accesses for a CL layer, first, we compute the off-chip memory accesses for one trip of $ifm$, $ofm$ and $wts$ using Algorithm 1 of Chapter 2 as below

$$\begin{aligned} \mathbb{B}_{ifm} =& \textbf{\textit{BWA}}(Addr_{ifm}, \langle T_{c_i}, T_{r_i}, T_{n_i} \rangle, \langle W_i, H_i, N_i \rangle, \delta) \\ \mathbb{B}_{ofm} =& \textbf{\textit{BWA}}(Addr_{ofm}, \langle T_{c_o}, T_{r_o}, T_{m_o} \rangle, \langle W_o, H_o, M_o \rangle, 0) \\ \mathbb{B}_{wts} =& M_o \cdot \textbf{\textit{BWA}}(Addr_{wts}, \langle K, K, T_{n_i} \rangle, \langle K, K, N_i \rangle, 0) \end{aligned} \tag{3.8}$$

where $\delta = (K - S)$ is the number of overlapping elements between adjacent $ifm$ tiles. $\mathbb{B}_{ifm}$, $\mathbb{B}_{ofm}$, and $\mathbb{B}_{wts}$ are the number of bytes accessed from off-chip memory for one trip of *ifm, ofm* and *wts* respectively. $\langle W_i, H_i, N_i \rangle$ are the *ifm* data shape and $\langle T_{c_i}, T_{r_i}, T_{n_i} \rangle$ are the *ifm* tile dimensions. *ofm* layer shape $\langle W_o, H_o, M_o \rangle$ is computed using Equation 3.3 and $ofm$ tile dimensions can be computed using Equation 3.3 as below,

$$\begin{aligned} T_{c_o} &= \frac{T_{c_i} - K}{S} + 1 \\ T_{r_o} &= \frac{T_{r_i} - K}{S} + 1, \end{aligned} \tag{3.9}$$

The total number of bytes accessed for j$^{th}$ reuse scheme ($\mathbb{B}_j$) can be expressed as following sum

$$\mathbb{B}_j = \mathbf{r_j} \cdot \begin{bmatrix} \mathbb{B}_{ifm} & \mathbb{B}_{ofm} & \mathbb{B}_{wts} \end{bmatrix}^T \tag{3.10}$$

where $\mathbf{r}_j$ is the row vector of the matrix $\mathbf{R}$ (Equation 3.7) for the j$^{th}$ scheme.

### 3.2.3 Optimization problem

Now, we present determining the optimal tile dimensions as a constraint optimization problem. Tiles of *ifm, ofm* and *wts* reside in on-chip memory. The volume of the tiles is given by equation Equation 3.11 below,

$$\begin{bmatrix} V_i \\ V_o \\ V_w \end{bmatrix} = \begin{bmatrix} T_{c_i} \cdot T_{r_i} \cdot T_{n_i} \\ T_{c_o} \cdot T_{r_o} \cdot T_{m_o} \\ K^2 \cdot T_{n_i} \cdot T_{m_o} \end{bmatrix} \tag{3.11}$$

where $V_i, V_o$, and $V_w$ are the sizes of *ifm, ofm*, and *wts* tiles, respectively. If the on-chip memory buffer size is $buffSize$ and each data element is represented by $DW$ bytes, then constraints on tile dimensions are

$$\begin{aligned} &(V_i + V_w + V_o) \cdot DW \leq buffSize \\ &0 < T_{c_o} \leq W_o, \;\; 0 < T_{r_o} \leq H_o \\ &0 < T_{n_i} \leq N_i, \;\; 0 < T_{m_o} \leq M_o \end{aligned} \tag{3.12}$$

Determining the tile dimensions which minimize the off-chip memory accesses, expressed by Equation 3.10, is a constraint optimization problem. The number of bytes accessed from off-chip memory using j$^{th}$ reuse scheme $\mathbb{B}_j$ ( Equation 3.10) and the constraints ( Equation 3.12) are non-linear functions of four variables $\langle T_{c_o}, T_{r_o}, T_{n_i}, T_{m_o} \rangle$, and thus solving it is non-trivial.

### 3.2.4 Off-chip memory access of FCL

The computations of FCLs are a special case of CLs with additional constraints on layer shapes and parameters. The *ifm* volume is same as a filter volume i.e., $H_i$=$W_i$=$K$, padding $P$=0, and stride $S$=1. In CNNs, typical values of K are 1, 3, 5, 7, and 11. Due to small values of $H_i$ and $W_i$, these dimensions are not partitioned ($T_{r_i}$=$T_{c_i}$=$K$). *ofm* layer shape and tile dimensions

computed using Equation 3.9 are $\langle 1, 1, M_o \rangle$ and $\langle 1, 1, T_{m_o} \rangle$.

For FCLs, the trips count of different data reuse schemes can be computed using Equation 3.7 and the number of bytes accessed from off-chip memory ($\mathbb{B}^{FC}$) using Equation 3.10 by applying the layer shape constraints. The constraints on tile dimensions are given by Equation 3.12. $T_{n_i}$ and $T_{m_o}$ are the unknowns to be determined in Equation 3.10. Determining the optimal tile dimensions for FCLs is a constraint optimization problem similar to CL.

## 3.3 Implementation and Results

### 3.3.1 Implementation

The optimal tile dimensions need to be determined once for a given NN and accelerator architecture, before the inference phase on edge devices. It can be determined by computing the number of bytes accessed from off-chip memory ($\mathbb{B}$) at all the feasible points in the solution space. We have developed a tool that determines the optimal tile dimensions of each NN layer during the *Preprocessing and Optimization* phase before the inference phase ( Figure 1.2 of Chapter 1). The tool first prunes the large search space of tiles by applying the constraints ( Equation 3.12). It then computes $\mathbb{B}$ using Equation 3.10 for each tile in the pruned search space and determines the optimal tile dimensions. The optimal tile dimension is determined for all the CL and FC layers of the NN. The tool also analyses the $\mathbb{B}$ for different data reuse schemes to suggest the best scheme for each CL. It can be configured for different on-chip memory sizes, bus widths, and data bit widths. It took few minutes to determine the optimal solution for VGG16 on Intel Core i7-6700 CPU (@3.40GHz$\times$8).

### 3.3.2 Validation

We have validated the number of bytes accessed from off-chip memory computed by BWA ( Algorithm 1 of Chapter 2) using the hardware implementation of CNN layers on Xilinx FPGA. We have implemented CNN layers using the Xilinx SDSoC framework, SDx v2018.3, which generates hardware functions from high-level languages like C/C++. We used the SDx pragmas to use zero_copy as a data mover. The Xilinx tools allow integrating the AXI Performance Monitor (APM) IP [52], which captures the real-time performance metrics like bus latency and amount of memory traffic for connected AXI interfaces. The target platform is ZedBoard, working at 100MHz frequency, and off-chip memory (DRAM) is accessed using 64 bits AXI bus. Our FPGA implementation takes the 3D shape and tile dimensions as input, and the generated hardware functions access the 3D data from DRAM using loop tiling. The integrated APM IP logs the number of bytes and latency of off-chip memory access transactions using which we validated the number of bytes accessed from off-chip memory computed by our approach for different 3D data shapes and tile dimensions.

### 3.3.3 Benchmarks

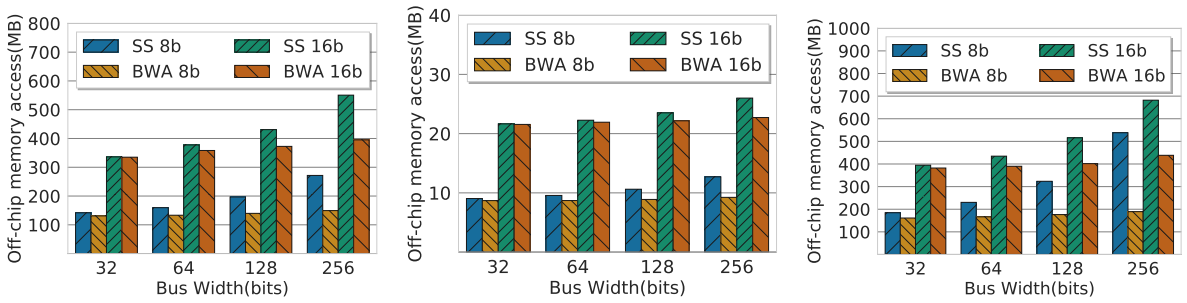
We carried out experiments on three popular CNN networks, VGG16 [44], AlexNet [26], and ResNet [22] having 8, 16, and 50 layers, respectively. These CNNs have varying shapes and use filters of dimensions $1\times1$, $3\times3$, $5\times5$, $7\times7$, and $11\times11$. To compare the results with other approaches, we have used the on-chip buffer size of 108 KB, batch size of 3 for VGG16, and 4 for ResNet and AlexNet, as used by Eyeriss [9] and SmartShuttle [28].

### 3.3.4 Baselines

We have implemented the SmartShuttle (SS) [28] approach to compare the results with our bus width aware (BWA) approach. SS statically determines the partitioning and scheduling scheme for each layer. It performs better than strategies that use a fixed tile dimension and data reuse

scheme for all layers, e.g., Eyeriss [9] and Zhang [55].

### 3.3.5 Results




(a) $\mathbb{B}_{VGG16}$ (b) $\mathbb{B}_{AlexNet}$ (c) $\mathbb{B}_{ResNet}$

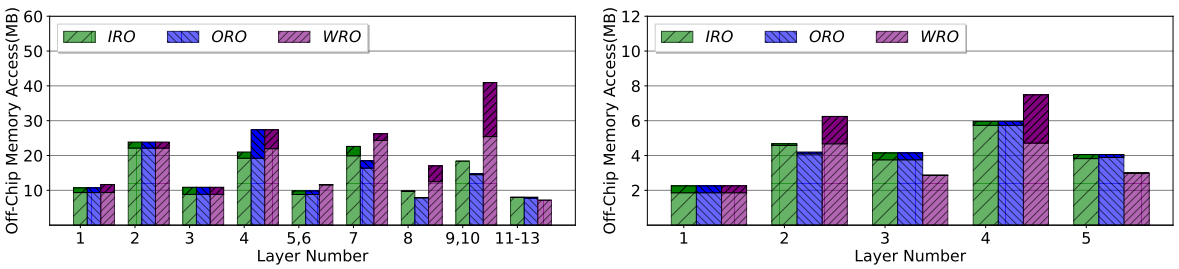
Figure 3.7: Off-chip memory access of convolution layers for 8 and 16 bits data width. BWA: Bus Width Aware, SS: SmartShuttle

#### 3.3.5.1 Impact of Bus Width on memory access of CLs

Figure 3.7 shows the number of bytes accessed from off-chip memory ($\mathbb{B}$) of CLs of the CNNs for different bus widths for 8 and 16 bits data width and 108 KB on-chip buffer size. Data is accessed in multiple of bus widths, and when the data length is not a multiple of the bus width, it leads to inefficient use of bus bandwidth. This inefficiency is more pronounced for wider data buses. For example, accessing 20 bytes on both 8-byte and 16-byte wide data buses results in 24 and 32 bytes of off-chip memory accesses, respectively. Consequently, the advantages of the BWA approach are more noticeable on wider data buses compared to narrower ones, as shown in Figure 3.7. The BWA approach reduces bus width's impact on off-chip memory accesses, as it selects the tile dimensions considering the bus width and address alignments. Whereas tile dimensions selected by SS remain the same, irrespective of the bus width of the accelerator, which results in a large value of $\mathbb{B}$. As shown in Figure 3.7, BWA reduces $\mathbb{B}$ compared to SS for the three CNNs. For ResNet:50 it reduces $\mathbb{B}_{ResNet}$ by 13%, 28%, 46%, and 65% for 8 bits data width and by 10%, 22% and 36% for 16 bits data width on 64, 128, and 256 bits wide data bus, respectively, compared to SS. BWA reduces $\mathbb{B}_{VGG16}$ by 8%, 16%, 29%, and 45% and

$\mathbb{B}_{AlexNet}$ by 4%, 9%, 16% and 27% on 32, 64, 128, and 256 bits wide buses respectively, for 8 bits data width. The impact of bus width is significant when accessing low-resolution data on a wide data bus. For 16 bits data width, the effectiveness of the BWA approach is noticeable for 64 or wider data buses. For 16 bits data width, BWA reduces $\mathbb{B}_{VGG16}$ by 5%, 13.5% and 28% and $\mathbb{B}_{AlexNet}$ by 1.5%, 5.7% and 13% compared to SS on 64, 128, and 256 bits wide data bus, respectively.

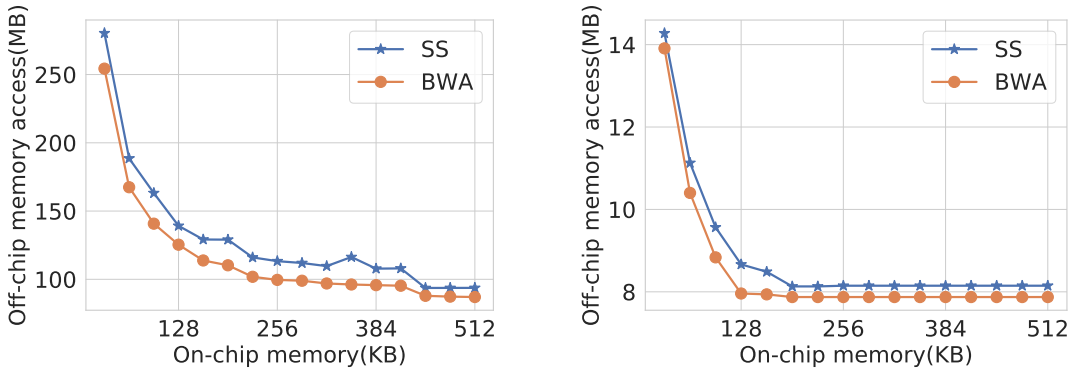
#### 3.3.5.2 Off-Chip Memory Access of Data Reuse Schemes




(a) VGG16 CLs (b) AlexNet CLs

Figure 3.8: Layer wise off-chip memory access for IRO, ORO and WRO schemes.

Figure 3.8 shows the layer-wise off-chip memory access of CLs of VGG16 and AlexNet for the three data reuse schemes using 64 bits wide bus and 8 bits data width. The results show that a single data reuse scheme is not optimal for all the layers. IRO and ORO perform better than the WRO scheme in the first few layers, while in the last few layers, WRO outperforms the other two schemes. The solid color at the top of the bars shows the reduction in off-chip memory accesses when optimal tile dimensions are selected using the BWA approach compared to when tile dimensions are selected using the tile size-based approach used by SS. For a data width of 8 bits, BWA achieves reductions of 7%, 11%, and 16% in $\mathbb{B}_{VGG16}$ and 34%, 34%, and 16% in $\mathbb{B}_{AlexNet}$ when compared to SS. These improvements are observed for IRO, ORO, and WRO reuse schemes, respectively, on a 64-bit-wide data bus.




(a) $\mathbb{B}_{VGG16}$ (b) $\mathbb{B}_{AlexNet}$

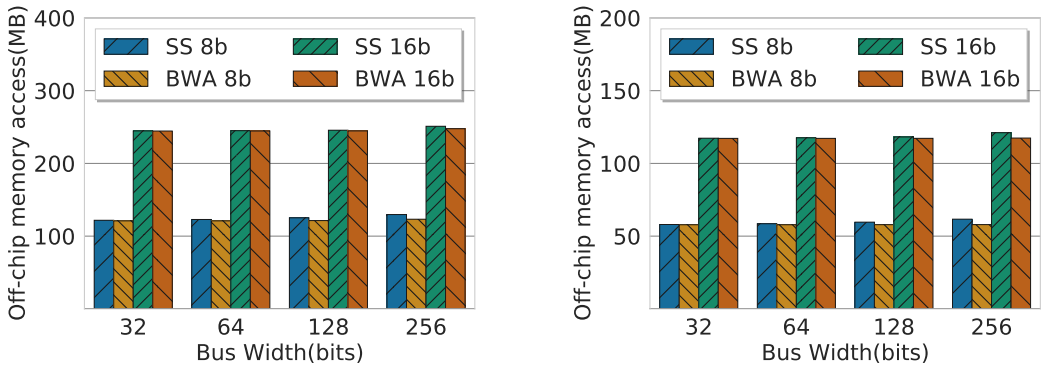
Figure 3.9: Off-chip memory access for varying on-chip buffer sizes. BWA: Bus Width Aware, SS:SmartShuttle

#### 3.3.5.3 On-chip Buffer Size

Figure 3.9a and Figure 3.9b compares the number of bytes accessed from off-chip memory ($\mathbb{B}$) for different on-chip buffer sizes ($buffSize$) for VGG16 and AlexNet, respectively. A large on-chip buffer can accommodate larger tiles, which reduces the tiles' trip counts ( Equation 3.7) and, therefore, the total off-chip memory accesses of the CNN ( Equation 3.10). This behavior is observed for both the CNNs in Figure 3.9a and Figure 3.9b. Both approaches select the dimensions of the tiles with the constraints of on-chip buffer size. However, the proposed BWA approach performs better as it considers address alignments and bus width to reduce unwanted data transfers and optimize off-chip memory accesses. In contrast, the SS approach ignores the architectural parameters.

#### 3.3.5.4 Impact of Bus Width on $\mathbb{B}$ of FCLs

In fully connected (FC) layers, spatial dimensions (height and width) are notably smaller compared to convolutional (CL) layers. For instance, the spatial dimensions of the last two FC layers in VGG16 are $1\times1$, whereas the spatial dimensions of the first two CL layers are $224\times224$. When tiling the spatial dimensions of CLs, a large number of unaligned accesses occur, unlike in FC layers where such unaligned accesses are absent. The absence of tiling in FC layers for spatial dimensions reduces the occurrence of unaligned accesses. Consequently,

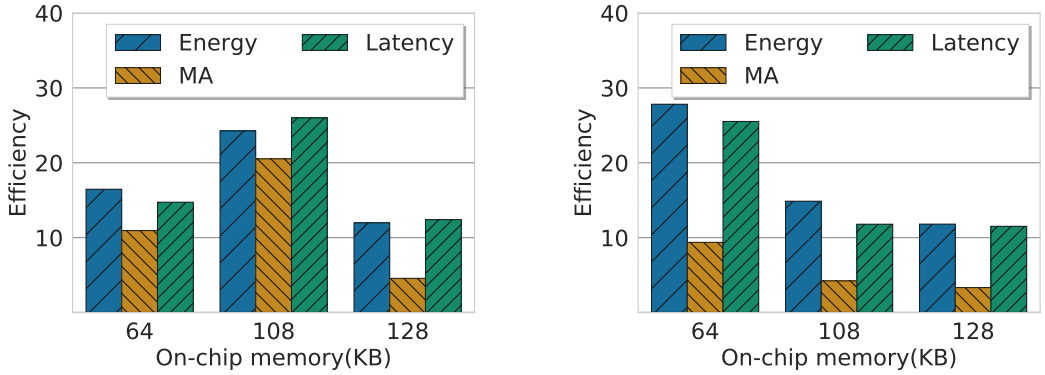



(a) VGG16 (b) AlexNet

Figure 3.10: Off-chip memory access of Fully connected layers. BWA: Bus Width Aware, SS:SmartShuttle

the advantages of the bandwidth (BW) approach on $\mathbb{B}$ are not as significant in FC layers compared to CL layers. Figure 3.10a and Figure 3.10b shows the off-chip memory accesses of FCLs of VGG16 and AlexNet, respectively, for 8 bits data width. BWA reduces $\mathbb{B}^{FC}_{VGG16}$ by 1%, 2%, 3%, and 4% and $\mathbb{B}^{FC}_{AlexNet}$ by 0.2%, 1%, 3% and 6% on 32, 64, 128, and 256 bits wide buses, respectively.

#### 3.3.5.5 Latency And Energy Analysis




(a) VGG16 (b) AlexNet

Figure 3.11: Energy and latency efficiency. BWA: Bus Width Aware, SS:SmartShuttle

Using our FPGA implementation, we measured the memory access latencies and execution time for CLs for the optimal tile dimensions determined using the approach described in Section 3.2. To estimate the energy efficiency achieved by the BWA compared to the SS

approach, we computed the energy consumption using the following equation [47]

$$E = P \cdot Time + \mathbb{B} \cdot E_{DDR} \tag{3.13}$$

where $P$ is the FPGA design power reported by the Vivado synthesis tool, $Time$ is the execution time, $\mathbb{B}$ is the number of bytes accessed from off-chip memory logged using Xilinx APM IP, and $E_{DDR}$ is the off-chip memory access energy per bit. We have used $E_{DDR}$=70 pJ/bit, a typical value for the DDR3 memory access energy [31].

Figure 3.11a and Figure 3.11b show the energy, off-chip memory accesses, and latency efficiency achieved using the BWA compared to the SS approach for VGG16 and AlexNet, respectively, for 8 bits data width and 64 bits bus width. We observed that the changes in energy and latency are proportional to the changes in memory access. This observation confirms that off-chip memory access dominates the energy consumption of the CNN accelerators.

## 3.4 Summary

Off-chip memory accesses dominate the energy consumption of CNN accelerators. Loop tiling is a common technique to partition the layer data into smaller tiles that fit into on-chip memory. The tile dimensions significantly impact the off-chip memory accesses of these accelerators. In this work, we propose a bus width and address alignment aware approach to compute the off-chip memory accesses of 3D data. Our tool statically analyses the memory accesses to find the optimal tile dimensions for CNN accelerators. Experimental results show that our approach reduces off-chip memory accesses of the CLs of VGG16 by 16%, 29%, and of AlexNet by 9%, 16% on 64 and 128 bits data bus, respectively, for 8 bits data width, compared to the state-of-the-art approach.

# Chapter 4

# Optimizing the Performance of RNN/LSTM Accelerators

## 4.1 Introduction

Many applications involve sequential data processing and time-series predictions, e.g., natural language processing, speech recognition, music composition, and video activity recognition. As convolution neural networks (CNNs) are specialized for processing image data, recurrent neural networks (RNNs) are specialized in handling sequential data. Processing sequential data requires remembering the contextual information from previous data. Recurrent neural networks (RNNs) are specialized in handling such problems by maintaining an internal state based on previously seen data. RNNs scale well with long sequences and even sequences of variable lengths. They share weights across different time steps. LSTMs [23] are variants of RNNs designed to handle long-range dependencies by storing useful information about previous inputs for a long duration.

LSTM computations involve several large matrix-vector multiplications, which are performed for many time steps. The inputs to the network are a time sequence of vectors, and these large matrices hold weights, which are learned during the training process. The sizes of

these matrices can be significant in several MBs and often exceed the size of the accelerator's on-chip memory. These matrices are partitioned into blocks and accessed from off-chip memory repeatedly by the accelerator, which results in a large volume of off-chip memory accesses and energy consumption.

## 4.2 Background

LSTM has recurrent connections to capture the long and short-term dependencies. LSTM cells maintain the cell state to store the dependency information derived from the previously seen data and use four gates to modify the cell state and produce the output. Typically the computations of LSTM cell is described by the following equations

$$\begin{aligned} i &= \sigma(W^i \cdot x_t + R^i \cdot h_{t-1} + b^i) \\ f &= \sigma(W^f \cdot x_t + R^f \cdot h_{t-1} + b^f) \\ g &= \tanh(W^g \cdot x_t + R^g \cdot h_{t-1} + b^g) \\ o &= \sigma(W^o \cdot x_t + R^o \cdot h_{t-1} + b^o) \\ c_t &= f \odot c_{t-1} + i \odot g \\ h_t &= o \odot \tanh(c_t) \end{aligned} \tag{4.1}$$

where $x_t$ is the input, $h_t$ is the hidden state and $c_t$ is the cell state at time $t$. $i, f, g, o$ are the computed gate values. $\odot$ denotes the element-wise multiplications. $W^j$ and $R^j$ are the input and hidden state weight matrices, respectively, and $b^j$ is the bias vector, where $j \in \{i, f, o, g\}$. $W^j$, $R^j$, and $b^j$ are the parameters learned during the training process. Once the network is trained, these parameters are used during inferencing. If the dimensions of the input vector $x_t$ is $L$ and that of the hidden state vector $h_t$ is $N$, the dimensions of $W^j$, $R^j$ and $b^j$ are $N \times L$, $N \times N$ and $N$, respectively. $N$ is the number of hidden states of the LSTM.

Equation 4.1 involves matrix-vector multiplications and element-wise operations. Element-

wise operations are vector-vector additions, multiplications, and non-linear functions. The non-linear functions are hyper-tangent ($\tanh$) and sigmoid ($\sigma$). LSTM computations ( Equation 4.1)

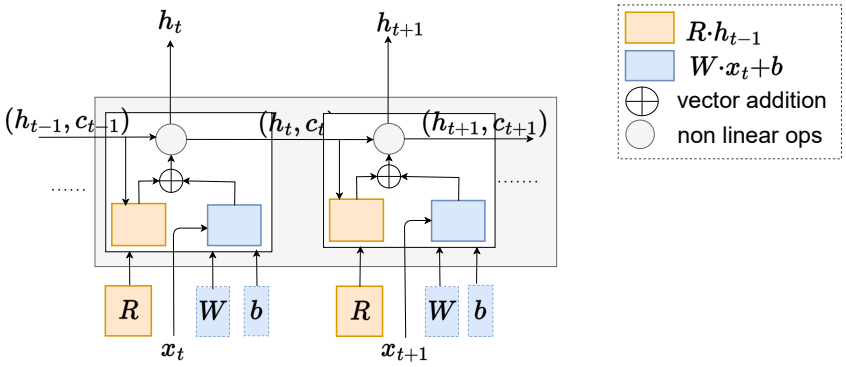



Figure 4.1: Data dependency in computations of LSTMs between consecutive time steps

for two consecutive time steps are shown in Figure 4.1. At every time step, Equation 4.1 take vectors $x_t$ as input and compute the cell state ($c_t$) and hidden state ($h_t$) using the previous hidden state ($h_{t-1}$) and cell state ($c_{t-1}$) vectors. $h_t$ depends on the present input vector $x_t$ and the previous time step cell state ($c_{t-1}$) and hidden state ($h_{t-1}$) vectors. The dependency of $h_t$ on $h_{t-1}$ and $c_{t-1}$ prevents the parallel processing of multiple time steps.

LSTM accelerators have small on-chip memory. The large weight matrices $R$ and $W$ are stored in off-chip memory. The dependency of $h_t$ and $c_t$ on the previous time step computations makes the reuse of weight matrix $R$ a challenge. Thus the weights are accessed from the off-chip memory at every step, resulting in sizeable off-chip memory accesses and high energy consumption of these accelerators. This work focuses on reducing the off-chip memory accesses of $R$ during the LSTM inference phase.

Our approach splits the computations of a time step and computes the partial sums of two consecutive time steps. As shown in Figure 4.2, the proposed approach at time step t computes $h_t$ using the input partial sum $S_t^U$, and partial sum computed at current time step $S_t^L$. $h_t$ is then used to compute the partial sum $S_{t+1}^L$ for next time step, while reusing the weights of $R$. $S_{t+1}^L$ is then passed for the next step computations of $h_{t+1}$. In our approach, only half of the matrix $R$ is accessed at each time step, reducing the $R$ matrix accesses by half.

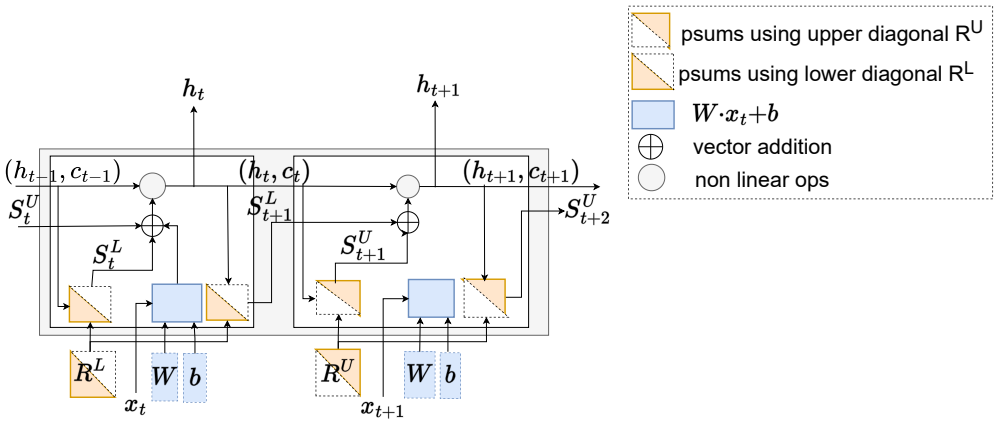



Figure 4.2: Proposed approach: Splitting computations into partial sums and reusing the weights of $R$.

## 4.3 Related Work

To address the computational and energy efficiency of LSTMs and, in general RNNs, several ASIC [4, 11, 49] and FPGA based accelerators [5, 14, 17, 20, 27] are proposed. The energy efficiency of LSTM accelerators is critical for their widespread usage, and off-chip memory access is the key to improving energy consumption. Most of these works focused on improving energy efficiency by reducing off-chip memory accesses.

Some approaches [14, 27, 42] used on-chip memory to store all the weights. Sizes of weights in recent multi-layer LSTM models can be several MB's, and using large on-chip memory is expensive. These approaches are not scalable and effective only for small LSTM models. Our approach is independent of model size and effective for large LSTM models.

Several approaches used the fact that neural networks are error-tolerant and have lots of redundancy. They used quantization and pruning techniques to compress the models' size. Approaches [14, 48] used 18-bit, Chang et al. [5] used 16-bit, Han et al. [20] used 12-bit precision for storing the inputs and weights, Lee et al. [27] used 8-bit inputs and 6-bits for weights to reduce the model size. Our approach is orthogonal to the quantization techniques and can be integrated with different quantization techniques to reduce memory access further.

Han et al. [20] used pruning to compress the model. However, pruning results in irregular

network structure and the sparse matrices require additional computational and storage resources resulting in unbalanced load distribution. To overcome this, Wang et al. [48] used block-circulant matrices representations to compress the LSTM/RNN model and eliminate irregularities resulting from compression. Some approaches [20, 35, 36] used load balance aware pruning techniques to overcome the unbalanced load distribution problem.

Quantization and pruning approaches compromise the accuracy of the networks. The other line of work reduced the memory accesses without affecting the accuracy of the NNs output by applying the data-reuse techniques. The matrix-vector multiplication $W^j \cdot x$ in Equation 4.1, where $j \in \{i, f, g, o\}$, is independent of previous state computation. Que et al. [41] proposed a blocking-batching scheme that reuses the weights of $W^j$ matrix by processing a group of input vectors as a batch. The input vectors in the same batch share the same weight matrices ($W^j$). However, it is difficult to collect the required number of input vectors. As the LSTM cell states ($h_t$ and $c_t$) computations depend on previous time-step cell states, the benefit of their batching schemes is limited to $W^j \cdot x$. Reusing weights of $R$ across different time steps has not been successful because of the dependency on previous time-step states.

Park et al.( [37]) proposed a time-step interleaved weight reuse scheme (TSI-WR) which reuses the weights of $R$ matrix between two adjacent time steps by performing computations in a time-interleaved manner. Their approach logically partitions the $R$ matrix into blocks. A block is accessed from off-chip memory to compute the hidden state vector $h_t$, and a fraction of it is reused to compute the partial sum of next time step state $h_{t+1}$. However, their approach only partially exploits the data reuse, and several weights are accessed repeatedly from the off-chip memory. In addition, the data reuse in the TSI-WR approach depends on the on-chip storage size, which benefits accelerators with larger on-chip memory.

Our approach schedules the computations in a way that reuses all the weights of $R$ between two adjacent time steps. The data reuse in our approach is independent of on-chip buffer sizes, which even benefits to accelerators with small on-chip memory.

## 4.4 Split And Combine Computations Approach

In this section, we first describe the basic idea of the Split And Combine Computations (**SACC**) approach and then its extension to block-wise data reuse.

### 4.4.1 Basic Approach

The computation of $h_t$ can be expressed as follows:

$$h_t[k] = F(S_t[k] + q_t[k]) \tag{4.2}$$

Where $F$ is a non-linear function, and $q_t$ is computed as $W \cdot x_t + b$. The computations of $q_t$ are independent of the previous step's cell states. $S_t[k]$ is the sum of $N$ product terms, given by:

$$S_t[k] = \sum_{n=0}^{N-1} R[k][n] \cdot h_{t-1}[n] \tag{4.3}$$

Computation of $S_t[k]$ (and, consequently, $h_t[k]$) depends on all $N$ elements of $h_{t-1}$. Traditional approaches compute all $N$ elements of $h_{t-1}$ first, requiring access to the full matrix $R$, before initiating computations for $h_t$. However, due to limited on-chip memory, elements of matrix $R$ are replaced by other elements required for $h_t$ computations, and when the accelerator starts computing $h_{t+1}$, the weights of $R$ are reaccessed from the off-chip memory.

To enable reuse of the weights of $R$, we split the computations of $S_t[k]$ into two partial sums as follows:

$$S_t[k] = \sum_{n=0}^{k} R[k][n] \cdot h_{t-1}[n] + \sum_{n=k+1}^{N-1} R[k][n] \cdot h_{t-1}[n] \tag{4.4}$$

We define $S_t^L[k]$ and $S_t^U[k]$ as the first and second partial sums in the above equation,

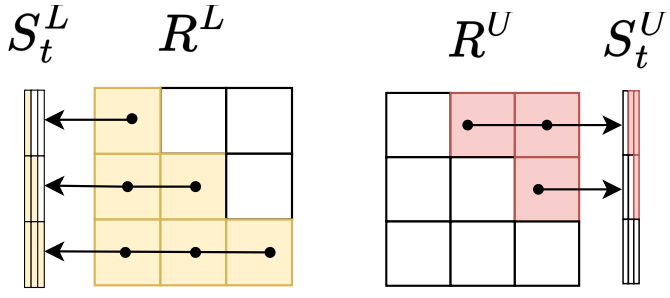
respectively. They are expressed as follows:

$$S_t^L[k] = \sum_{n=0}^{k} R[k][n] \cdot h_{t-1}[n] \tag{4.5}$$

$$S_t^U[k] = \sum_{n=k+1}^{N-1} R[k][n] \cdot h_{t-1}[n] \tag{4.6}$$

Here, $S_t^L[k]$ uses the lower-diagonal and diagonal elements of $R$ ($R^L$), while $S_t^U[k]$ uses the upper-diagonal elements of $R$ ($R^U$). The elements of $R$ used in the computations of $S_t$ are also required for the partial sum computations of $S_{t+1}$. In our approach $S_{t+1}^L[k]$ computations can be initated soon after first $k+1$ elements of $h_t$ are computed. Similarly at the next time step, the computations of $S_{t+2}^U[k]$ can be initiated soon after the last $N-(k+1)$ elements of $h_{t+1}$ are computed. By dividing the computations into two partial sums, we enable the reuse of weights of $R$ between two consecutive time steps. Our approach accesses the weights of $R$ and reuses them for computations of partial sums for two consecutive time steps.

In Figure 4.3a and Figure 4.3b, we can observe the elements of matrix $R$ that are utilized in the partial sum computations of $S_t^L$ and $S_t^U$, respectively, for an example $3 \times 3$ matrix $R$. In this case, each element of the resulting vector $S$ is the sum of three product terms. Notably, the partial sum vectors $S_t^L$ and $S_t^U$ only contain a fraction of the total sum $S$, and these fractions are visually highlighted using colored rectangles in Figure 4.3.




(a) (b)

Figure 4.3: Fraction of partial sums computated using (a) lower diagonal element of $R$ (b) upper diagonal elements of $R$

An illustrative example that demonstrates the computations performed in two consecutive steps, $t$ and $t+1$, is presented in Figure 4.4 and Figure 4.5 for a $3\times3$ matrix $R$. The bold lines in Figure 4.4 and Figure 4.5 highlight the reuse of weights from matrix $R$ at different stages.

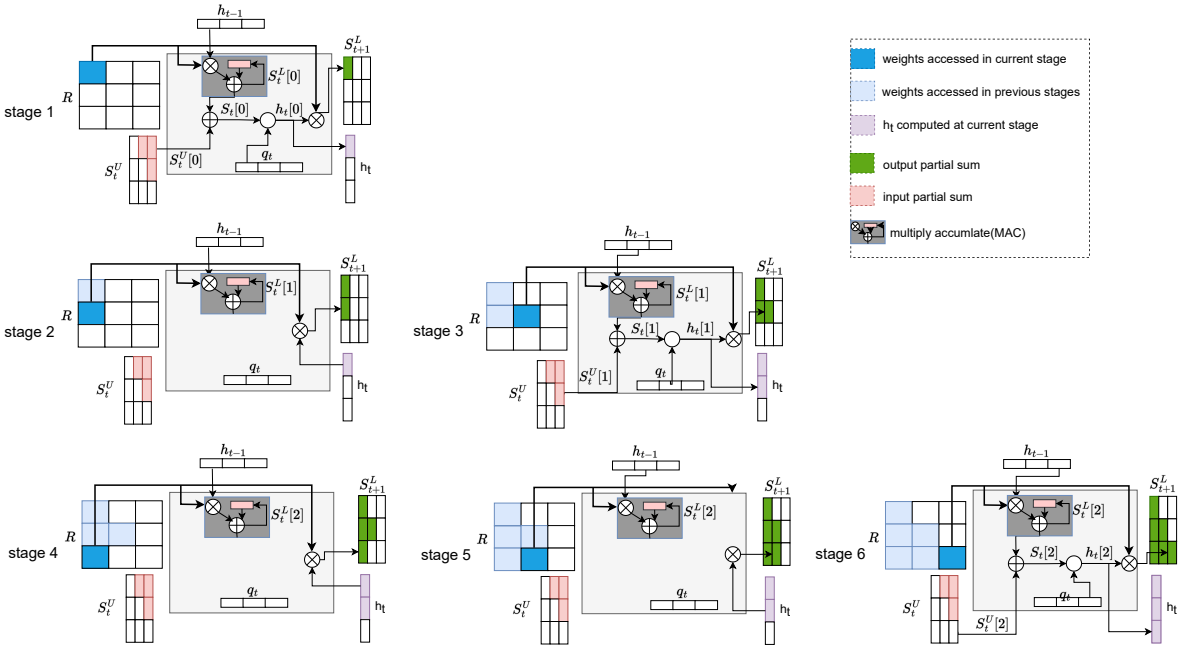



Figure 4.4: Computation stages at time step $t$.

At time step $t$, $R^L$ is accessed and there are $\frac{N\times(N+1)}{2}$ stages. In this specific example, with $N = 3$, we have six stages at time step $t$. In each stage, an element of matrix $R$ is accessed and processed through a Multiply Accumulate Unit (MAC). The processing involves multiplying the element of $R$ with a corresponding element from the vector $h_{t-1}$, and the product is accumulated. This process continues until all the elements of a row of matrix $R^L$ have been accessed. Subsequently, the output of the MAC is added to the corresponding element of vector $S_t^U$ to compute the element of vector $h_t$, and the MAC unit is then reset to zero. Furthermore, the elements of matrix $R^L$ are efficiently reused to compute the partial sum vector $S_{t+1}^L$, utilizing the values of vector $h_t$ computed in previous stages or current stage.

Figure 4.5 illustrates the computation stages for time step $t+1$, which utilizes the previously computed $S_{t+1}^L$ from time step $t$. At this particular time step, there are $\frac{N\times(N-1)}{2}$ stages. In this

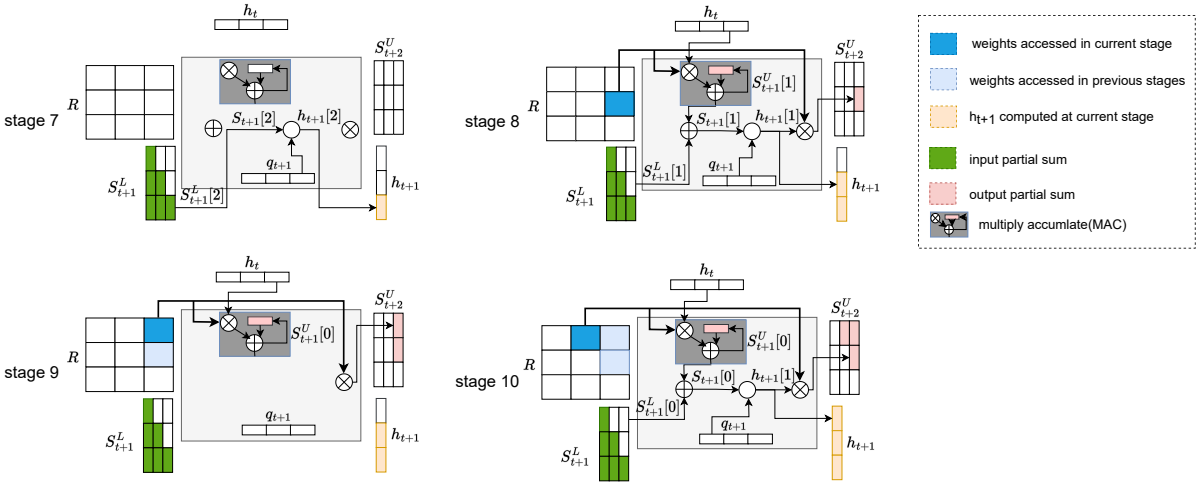



Figure 4.5: Computation stages at time step $t+1$.

example, with $N = 3$, we observe three stages at time step $t+1$. For each stage, the inputs are $h_t$ and $S^L_{t+1}$, which are carried over from the previous time step $t$. Subsequently, the computation results in the outputs $h_{t+1}$ and the partial sum vector $S^U_{t+2}$, efficiently reusing the weights of matrix $R^U$.

By analyzing both Figure 4.4 and Figure 4.5, we can observe that all the elements of matrix $R$ are accessed once, enabling the computation of the outputs $h_t$ and $h_{t+1}$ along with the partial sum $S^U_{t+2}$, which is then passed on to the next time step.

### 4.4.2 Block-wise reuse

The basic approach accesses one element of $R$ at each stage. However, to leverage the LSTM accelerator's capability of storing multiple elements of $R$ and processing multiple elements simultaneously, the basic approach is extended by partitioning $R$ into square matrices. The accelerator's on-chip memory determines these square matrices' size. Specifically, let $R$ be partitioned into square matrices $P$ of size $B \times B$, where $B$ is chosen such that $P$ fits into the on-chip memory. Each $P$ matrix is indexed as $(r, m)$, where $0 \leq r, m \leq (\lceil \frac{N}{B} \rceil - 1)$, as illustrated in Figure 4.6a.

These small matrices are then grouped into two sets, $R^L$ and $R^U$. The set $R^L$ contains all

the matrices with index $r \geq m$, while $R^U$ contains all the matrices with index $r < m$, as depicted in Figure 4.6b and Figure 4.6c, respectively.

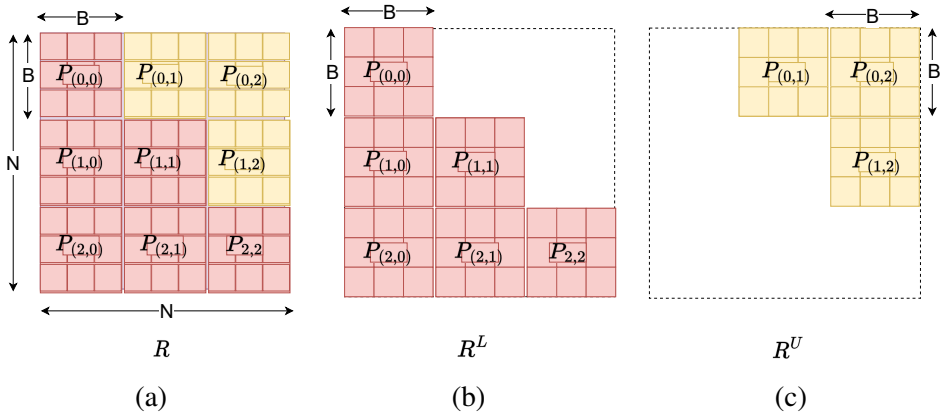



Figure 4.6: (a) Partitions of a matrix $R_{N \times N}$ into $B \times B$ square matrices. (b) $R^L = \{P_{(r,m)} : r \geq m\}$ (b) $R^U = \{P_{(r,m)} : r < m\}$

In the block approach, computations of $h_t$ are divided into $\lceil \frac{N}{B} \rceil$ slices, each containing $B$ elements. For computing the $k^{th}$ element of the $r^{th}$ slice of $h_t$, where $0 \leq k \leq B-1$, the following equation is employed:

$$h_t[B*r+k] = F\Big(\sum_{m=0}^{\lceil \frac{N}{B} \rceil - 1} S_{(r,m)}[k] + q_t[B*r+k]\Big) \tag{4.7}$$

where $S_{(r,m)}$ represents the partial sum of the $r^{th}$ slice of $S$, computed using the partition $P_{(r,m)}$. The element $S_{(r,m)}[k]$ is calculated using the equation below:

$$\begin{aligned} S_{(r,m)}[k] &= \sum_{j=0}^{B-1} P_{(r,m)}[k][j] \cdot h_{t-1}[B*m+j] \\ &= \sum_{j=0}^{B-1} R[B*r+k][B*m+j] \cdot h_{t-1}[B*m+j] \end{aligned} \tag{4.8}$$

The summation $\sum_{m=0}^{\lceil \frac{N}{B} \rceil -1} S_{(r,m)}[k]$ in Equation 4.7 can be broken down into two partial sums:

$$\sum_{m=0}^{\lceil \frac{N}{B} \rceil -1} S_{(r,m)}[k] = \sum_{m=0}^{r} S^L_{(r,m)}[k] + \sum_{m=r+1}^{\lceil \frac{N}{B} \rceil -1} S^U_{(r,m)}[k] \tag{4.9}$$

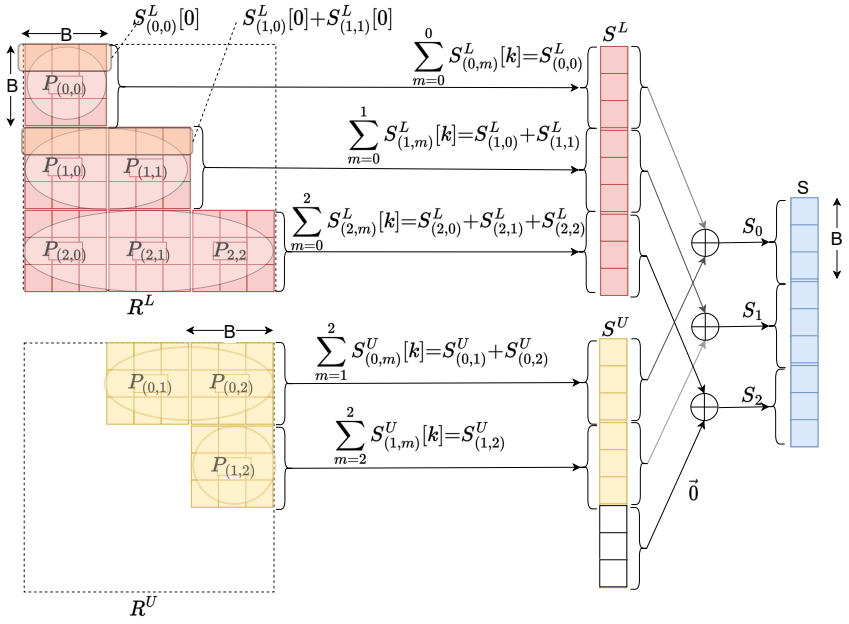



Figure 4.7: Partitions of $R^L$ and $R^U$ required for computations of partial sum $S^L$ and $S^U$.

Figure 4.7 further illustrates the partitions $P$ used in computing $\sum_{m=0}^{r} S^L_{(r,m)}[k]$ and $\sum_{m=r+1}^{\lceil \frac{N}{B} \rceil -1} S^U_{(r,m)}[k]$. Partial sum computations of the $r^{th}$ slice of the $(t+1)^{th}$ time step can be initiated as soon as the first $r$ slices ($B \times r$ elements) of $h_t$ are computed. Computations of the $r^{th}$ slice of partial sums ($S_L$ and $S_U$) at different time steps use the same partitions $P$ of $R$. The proposed approach exploits this to effectively reuse the weights of $P$ to compute the partial sums of two consecutive time steps. This optimization ensures efficient data dependency handling and speeds up the overall computation process.

Algorithm 2 describes the block wise SACC approach. The dimensions of $W$, $R$, $b$, and $x_t$ are $4N \times L$, $4N \times N$, $4N \times 1$, and $L \times 1$, respectively. The algorithm stores the vectors $h_t$, $c_t$, and the partial sum vectors ($s_{t+1}$) in the on-chip memory and accesses the weights from

the off-chip memory. It first computes the vector $q_t$ as $W \cdot x + b$, at line 2 and then invokes the procedures UPDIAGREUSE at line 4 or LOWDIAGREUSE at line 7 at alternate time steps. LOWDIAGREUSE accesses blocks of $R^L$, and UPDIAGREUSE accesses blocks of $R^U$. The procedures have two nested loops. LOWDIAGREUSE traverses the blocks from top to bottom (at line 11), left to the right (at line 13), while the UPDIAGREUSE traverses the blocks in the opposite order. The inner loop accesses the $(r, m)^{th}$ block of $R$ from the off-chip memory and reuses it to compute the partial sums $s^B_{t+1}$ and $s^B_{t+2}$. The outer loop iterations compute $r^{th}$ slices of $h_{t+1}$, $c_{t+1}$, and $s_{t+2}$. When all the blocks of the $r^{th}$ row are processed, $s^B_{t+1}$ has the total sum, which is then used to compute the $r^{th}$ slice of the vectors $h_{t+1}$ and $c_{t+1}$ using LSTMEQUATIONS at line 19. Both the procedures reuse the blocks of $R$ to reduce the off-chip memory accesses. Algorithm 3 implements the LSTM equations using the partial sum vector.

**Algorithm 2** SACC algorithm

**procedure** COMPLSTMCELL($W$,$R$,$b$,$x_{t+1}$)
  $q_{t+1} \leftarrow \text{MxV}(W, x_{t+1}, 4N, L)+b$
  **if** ($stage$ is even) **then**
    $(h_t, c_t, s_{t+1}) \leftarrow$UPDIAREUSE$(R, q_t, h_t, c_t, s_{t+1})$
    $stage \leftarrow odd$
  **else**
    $(h_t, c_t, s_{t+1}) \leftarrow$LOWDIAREUSE$(R, q_t, h_t, c_t, s_{t+1})$
    $stage \leftarrow even$
  **return** $(h_t)$
**procedure** LOWDIAGREUSE($R, q_t, h_t, c_t, s_{t+1}$)
  **for** $r \leftarrow 0$ **to** $(\lceil \frac{N}{B} \rceil - 1)$ **do**
    $(i_s, i_e) \leftarrow (r \cdot B, (r+1) \cdot B - 1), s^B_{t+2} \leftarrow \vec{0}$
    **for** $m \leftarrow 0$ **to** $r$ **do**
      $\mathbf{R}^B \leftarrow$GETDDRBLKS$(R, r, m, B)$
      $(h^B_t, c^B_t, q^B_t) \leftarrow$GETSLICE$(h_t, c_t, q_t, m)$
      $s^B_{t+1} \leftarrow$GETSLICES$(s_{t+1}, m)$
      $s^B_{t+1} \leftarrow s^B_{t+1} + \text{MxV}(\mathbf{R}^B, h^B_t, 4B, B)$
      **if** $m = r$ **then**
        $(h^B_{t+1}, c^B_{t+1}) \leftarrow$LSTMEQNS$(v^B_x, s^B_{t+1}, c^B_t)$
        $h_{t+1}[i_s : i_e] \leftarrow h^B_{t+1}, c_{t+1}[i_s : i_e] \leftarrow c^B_{t+1}$
      $h^B_{t+1} \leftarrow h_{t+1}[m \times B : (m+1) \times B - 1]$
      $s^B_{t+2} \leftarrow s^B_{t+2} + \text{MxV}(\mathbf{R}^B, h^B_{t+1}, 4B, B)$
    $s_{t+2} \leftarrow$UPDATEVECT$(s_{t+2}, s^B_{t+2}, r)$
  **return** $(s_{t+2}, h_{t+1}, c_{t+1})$
**procedure** UPDIAGREUSE($R, q_t, h_t, c_t, s_{t+1}$)
  **for** $r \leftarrow (\lceil \frac{N}{B} \rceil - 1)$ **downto** $0$ **do**
    $(i_s, i_e) \leftarrow (r \cdot B, (r+1) \cdot B - 1), s^B_{t+2} \leftarrow \vec{0}$
    **for** $m \leftarrow (\lceil \frac{N}{B} \rceil) - 1$ **downto** $r+1$ **do**
      $\mathbf{R}^B \leftarrow$GETDDRBLKS$(R, r, m, B)$
      $(h^B_t, c^B_t, q^B_t) \leftarrow$GETSLICE$(h_t, c_t, q_t, m)$
      $s^B_{t+1} \leftarrow$GETSLICES$(s_{t+1}, m)$
      $s^B_{t+1} \leftarrow s^B_{t+1} + \text{MxV}(\mathbf{R}^B, h^B_t, 4B, B)$
      $h^B_{t+1} \leftarrow h_{t+1}[i_s : i_e]$
      $s^B_{t+2} \leftarrow s^B_{t+2} + \text{MxV}(\mathbf{R}^B, h^B_{t+1}, 4B, B)$;
    $(h^B_{t+1}, c^B_{t+1}) \leftarrow$LSTMEQNS$(v^B_x, s^B_{t+1}, c^B_t)$
    $h_{t+1}[i_s : i_e] \leftarrow h^B_{t+1}, c_{t+1}[i_s : i_e] \leftarrow c^B_{t+1}$
    $s_{t+2} \leftarrow$UPDATEVECT$(s_{t+2}, s^B_{t+2}, r)$
  **return** $(s_{t+2}, h_{t+1}, c_{t+1})$

**Algorithm 3** LSTM Equations

**procedure** LSTMEQNS($q_{t+1}, s_{t+1}, c_t$)
  $(v_i, v_f, v_g, v_o) \leftarrow ExtractVecs(q_t, B)$
  $(s_i, s_f, s_g, s_o) \leftarrow ExtractVecs(s_{t+1}, B)$
  **for** $n \leftarrow 0$ **to** $B-1$ **do**
    $i[n] \leftarrow \text{SIGMOID}(s_i[n] + v_i[n])$
    $f[n] \leftarrow \text{SIGMOID}(s_f[n] + v_f[n])$
    $g[n] \leftarrow \text{TANH}(s_g[n] + v_g[n])$
    $o[n] \leftarrow \text{SIGMOID}(s_o[n] + v_o[n])$
    $c_{t+1}[n] \leftarrow f[n] \otimes c_t[n] + i[n] \otimes g[n]$
    $h_{t+1}[n] \leftarrow o[n] \otimes \text{TANH}(c)[n]$
    **return**$(h_{t+1}, c_{t+1})$

## 4.5 Experimental Setup And Results

We have implemented LSTM layers using the conventional, TSIWR [37], and our approach and synthesized the design using the SDSoC framework, SDx v2018.3. Figure 4.8 shows the main components and interfaces of the FPGA design. The design reads $B \times B$ size blocks of the four input $R$ matrices ($R_i, R_f, R_g$, and $R_o$) into the local buffers using bufferCopy component and then performs the matrix-vector multiplications using $M \times V$ component. The blocks are read in the order specified by the schedule. The schedule for all three approaches is prepared offline. The top-level logic first reads the blocks of $R$ matrices ($R_i, R_f, R_g$, and $R_o$) and invokes compute $M \times V$ to compute the four matrix-vector multiplications in parallel. The design uses the dual buffering scheme to overlap the computations with data transfer. The four matrix multiplication units ($M \times V_i, M \times V_f, M \times V_g$, and $M \times V_o$) perform matrix-vector multiplications in parallel, and the amount of parallelism in each $M \times V$ unit can be configured.

The design can be configured for different on-chip buffer sizes, hidden units ($N$), and compute resources. We have implemented multiple designs for each of the three approaches to compare the throughput for different numbers of compute resources and buffer sizes. We carried out the experiments on Zedboard, and the target frequency is 100MHz. The off-chip memory is DDR3 connected using a 64-bit AXI bus. We have integrated the Xilinx AXI

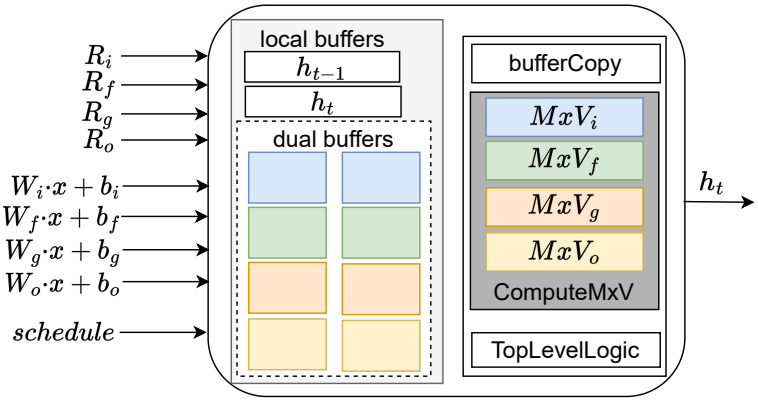



Figure 4.8: Block level FPGA Design for LSTM Implementation.

Performance Monitor (APM) IP to log the number of bytes transferred and memory access latencies for DRAM accesses.

There are eight local buffers each to store the $B \times B$ elements of the matrices (Figure 4.8), and two buffers, each of dimension $N$, for storing the vector $h_t$ and $h_{t+1}$. For TSIWR and our approach, additonal on-chip storage of size $4 \times N$ is required to store the partial sums of next time steps compared to conventional approach. Since the partial sums are stored in the on-chip memory, there is no additional impact on the latency compared to the conventional approaches. Table 4.1 shows the FPGA resources utilization reported by Vivado for conventional, TSIWR, and our approach, for $B$=64 and using 8 DSPs resources ($PARL\_FACTOR$=8) in each of the four $M \times V$ component. In this case, the total number of DSPs required is $4 \times PARL\_FACTOR$=32. Each of the eight arrays is partitioned into 8 ($PARL\_FACTOR$) to provide the data to compute resources. That results in 64 FPGA RAMB18 resources for storing the $B \times B$ blocks and 2 FPGA RAMB18 resources for storing the vector $h_t$ and $h_{t+1}$. The remaining RAMB18 blocks are used for other logic and vary for the three approaches. The conventional approach has the most straightforward control logic than TSIWR and our approach. The LUTs and FFs in the case of TSIWR and our approach are higher than the conventional approach.

Table 4.1: FPGA resource utilization for B=64, PARL_FACTOR=8, and N=128

| | RAMB18 | DSPs | LUTs | FFs |
|---|---|---|---|---|
| Conv. | 86 | 32 | 14107 | 17768 |
| TSIWR | 86 | 32 | 16669 | 22465 |
| SACC | 86 | 32 | 15738 | 18571 |

### 4.5.1 Baseline

We have compared our approach (SACC) with conventional and TSI-WR [37] approaches. In our experiments, to perform a fair comparison, the size of the on-chip memory, matrix multiplication logic, and number of compute resources are kept the same for all three approaches.

We have also implemented the models to compute the off-chip memory accesses for all three approaches and integrated them with the analytical framework (Chapter 2) to compare the off-chip memory accesses of the three approaches.

### 4.5.2 Benchmarks

To demonstrate the efficiency of our approach, we have experimented with LSTM models used in speech recognition (for TIMIT [15]) and character level Language Modelling (LM) [45]. These models are widely used in natural language processing. The LSTM models are adopted from [4, 20, 35]. Each model has two LSTM layers, and Table 4.2 list the parameters of these models.

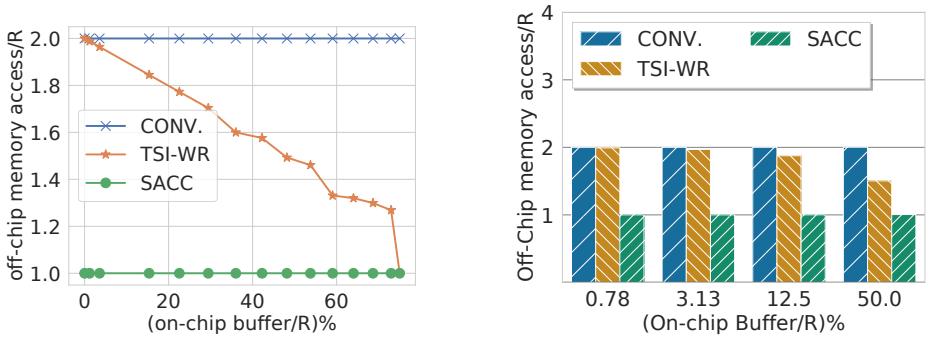
Table 4.2: LSTM Models used for experiments

| **Model** | **Input Size** | **#Hidden units** | |
|---|---|---|---|
| | | Layer 1 | Layer 2 |
| Character Level LM [4] | 65 | 128 | 128 |
| TIMIT-512 [35] | 40 | 512 | 512 |
| TIMIT-1024 [20] | 160 | 1024 | 1024 |

### 4.5.3 Results

#### 4.5.3.1 Off-Chip Memory Access

Using our FPGA designs (Figure 4.8) that can be configured for different input vector lengths, on-chip buffer sizes, and the number of hidden units we measured the off-chip memory accesses for conventional, TSI-WR and our SACC approach. The hardware implementation was integrated with Xilinx AXI Performance Monitor (APM) IP to log the number of bytes transferred between DDR3 and FPGA design. We have experimented with different on-chip memory sizes smaller than the size of $R$. Figure 4.9 shows the off-chip memory accesses for two consecutive time steps for conventional, TSI-WR, and SACC approaches. Figure 4.9a shows the results computed using Equation 2.7 of chapter 2, and Figure 4.9b shows the results measured using our hardware implementation on FPGA (Figure 4.8).




(a) (b)

Figure 4.9: Off-chip memory access for matrix-vector multiplication of two consecutive time steps with different on-chip buffer to R matrix size ratio. (a) using analytical framework (b) measured on hardware

If the on-chip buffer size is small compared to the weight matrices, tiles of $R$ are accessed from off-chip memory, replacing the older tiles in the on-chip memory. In conventional approaches, these tiles are not reused for subsequent time step computations, which results in accessing full matrix $R$ every step. For conventional approaches, the off-chip memory accesses remain the same, even if the on-chip mem size is increased, as shown by the horizontal line in Figure 4.9a. The TSI-WR approach schedules the tiles to reuse the data from

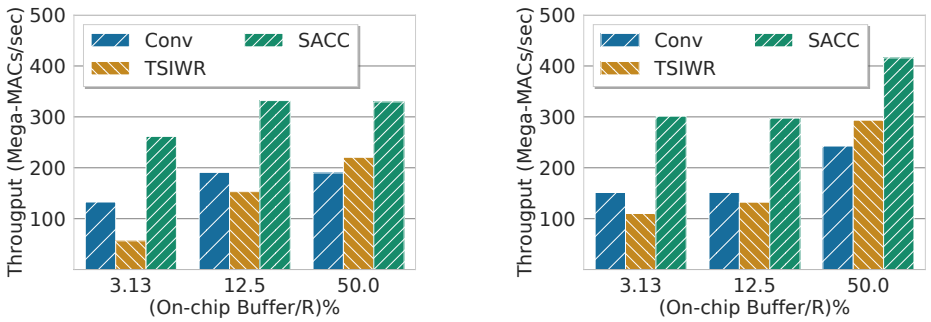
the on-chip memory and reduces off-chip memory access. However, the extent of data reuse in the TSI-WR approach depends on the size of overlap between two consecutive tiles, which is decided by the available on-chip buffer size. When the on-chip buffer to R matrix size is close to 48%, TSI-WR reduces the memory access by ≈25%, as shown in Figure 4.9. Our approach splits the computations and performs the scheduling of the tiles that reduces the memory accesses by ≈50%, irrespective of the on-chip buffer size.

#### 4.5.3.2 Throughput Analysis

To analyze the throughput variation with different on-chip buffer sizes, we synthesized FPGA designs (Figure 4.8) with varying buffer sizes for all three approaches. We recorded the number of cycles (measured_clock_cycles) for different FPGA designs over $T$ time steps to evaluate the throughput. Using this data, we computed the throughput using the following formula:

$$\text{Throughput(Mega-MACs/sec)} = \frac{4 \times N^2 \times T \times \text{clock\_freq}}{\text{measured\_clock\_cycles}} \tag{4.10}$$

Here, clock_freq represents the system clock frequency in MHz. Figure 4.10a and Figure 4.10b




(a) 64KB on-chip buffer size (b) 128 KB on-chip buffer size

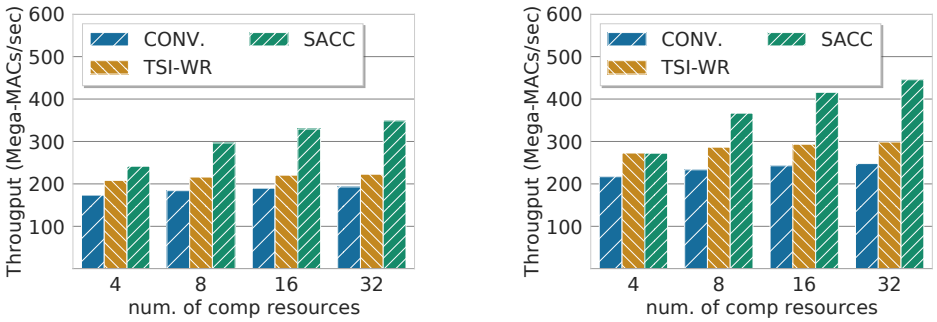
Figure 4.10: Throughput variation with different on-chip buffer/R ratio

compare the throughput for different ratios of on-chip buffer to weight matrix size for 64 and 128 KB on-chip buffer size, respectively. We experimented with different numbers of hidden units ($N$) to vary the $R$ matrix size. Increasing the on-chip buffer to $R$ matrix size ratio results

in larger tile sizes and fewer iterations, improving throughput for all three approaches. The TSI-WR approach performs worse than the conventional approach for smaller on-chip buffer/$R$ ratio due to its control logic overhead which overshadows the data reuse benefits. However, for a larger on-chip buffer/$R$ size ratio (e.g., 50%), the TSI-WR approach outperforms the conventional approach due to better data reuse. Our approach performs better than the other two for all the on-chip buffer to $R$ matrix size ratio due to its significant data-reuse of weight matrix $R$ and small control logic overhead. For 50% on-chip buffer to matrix size ratio, our approach improves the throughput by 42% and 33% for 64 KB and 41% and 29% for 128 KB on-chip buffer size, compared to conventional and TSI-WR approaches, respectively.

#### 4.5.3.3 Throughput variation with compute resources

The memory bandwidth limits the performance of LSTM accelerators. Data reuse reduces the memory bandwidth bottleneck. To observe the impact of data reuse on memory bandwidth, we have implemented multiple FPGA designs with different numbers of compute resources for each of the three approaches. Throughput is measured using Equation 4.10 for each approach. We have compared the results when on-chip buffer to matrix size ratio of 50%.




(a) 64 KB on-chip buffer size (b) 128 KB on-chip buffer size

Figure 4.11: Throughput variation with compute resources for 50% on-chip buffer/$R$ ratio

Figure 4.11 compares the throughput of the conventional, TSI-WR and our approach. Increasing the number of computing resources does not improve the performance of the conventional approach. TSI-WR approach improves the performance with increasing compute

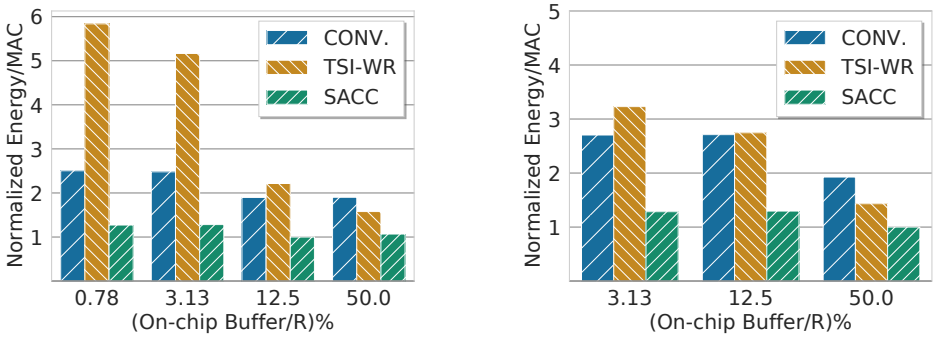
resources as it reuses the on-chip data and improves the operational intensity (operations/byte). However, the improvement in the TSI-WR approach is noticeable only for the large on-chip buffer to $R$ ratios. Our approach (SACC) has better operational intensity (operations/byte). It alleviates the memory bandwidth issue compared to the other two approaches, which improves throughput by increasing the number of parallel resources. The improvement in throughput with the increased number of compute resources using our approach is observed for different on-chip buffer sizes as shown in Figure 4.11a and Figure 4.11b.

#### 4.5.3.4 Energy Efficiency Improvement

We used our FPGA implementations to measure the run time for conventional, TSI-WR, and SACC approaches for many time steps. We computed the energy consumption of the designs using the following equation [47]

$$E = P \cdot Time + \mathbb{B} \cdot E_{DDR} \tag{4.11}$$

where $P$ is the FPGA design power reported by the Vivado synthesis tool, $Time$ is the average run time of the design, $\mathbb{B}$ is the number of bytes accessed from off-chip memory logged using Xilinx APM IP, and $E_{DDR}$ is the off-chip memory access energy per bit. We have used $E_{DDR}$=70 pJ/bit, a typical value for the DDR3 memory access energy [31]. Figure 4.12a and




(a) 64KB on-chip buffer size (b) 128 KB on-chip buffer size

Figure 4.12: Energy improvement with on-chip buffer size/R

Figure 4.12b show the normalized energy efficiency per MAC operation for different on-chip buffer to R matrix size ratios for 64 KB and 128 KB on-chip buffer sizes, respectively. All three approaches observe the improvement with the increase in the on-chip buffer size to matrix size ratio due to a reduction in the control logic execution. TSI-WR performs better than the conventional approach only after the on-chip buffer to R matrix size ratio is greater than 12.5%. Our approach (SACC ) outperforms the other two approaches for all the cases as it reduces the off-chip memory accesses, which dominates the overall energy consumption. For 50% on-chip buffer to matrix size ratio, the SACC approach reduces 43% and 32% energy for 64 KB on-chip buffer and 48% and 30% for 128 KB on-chip buffer size compared to conventional and TSI-WR approach, respectively.

## 4.6 Summary

Long Short-Term Memory (LSTM) networks are widely used in speech recognition and natural language processing (NLP). With the enormous growth in the number of Edge AI applications, there is a pressing need for efficient execution of these algorithms on edge devices. These edge devices use customized accelerators to meet energy and throughput targets. The key to improving the energy efficiency and throughput of DNN accelerators is to reduce the off-chip memory accesses. This work proposes a novel data reuse approach that reduces the off-chip memory accesses of large weight matrices of RNNs/LSTMs by $\approx$50% and improves the throughput significantly. Our approach improves the throughput by 55% and 29% and reduces the energy consumption by 52% and 30% for 12.5% and 50% on-chip buffer to matrix size ratio, for 128 KB on-chip buffer size, compared to state-of-the-art TSI-WR approach.

# Chapter 5

# Performance Improvement of SOM by using Low Bit-Width Resolution [1]

One of the prime reasons for NN's popularity in the last decade is their ability to detect patterns accurately. Compared to conventional solutions, they scale exceptionally well because of their inherent parallelism. The main strength of the NNs is their ability to learn the classification criteria directly from the input data. In essence, a NN trained with genomic data compresses its inputs and encodes it in its structure. The NN then can act as a predictor that can be queried about specific features in a given genome instead of attempting to output linear DNA sequences as done in the conventional assembly. In other words, NNs can invent an algorithm automatically that otherwise would have to be developed by bioinformatics engineers. The parameters of the networks define the algorithm as they are developed during its training. That also provides the possibility of re-training the network to update the identification features, such as mutations and other genomic alterations that were not known before, without changing the algorithm.

Yang et al. [53] has introduced Self-organizing maps (SOM) for rapid genome

[1] The experiments in this chapter involved joint efforts with other students at KTH Royal Institute of Technology, Sweden. Specifically, some parts of the research utilized measurement data obtained by these collaborators.

identification. SOM uses a type of unsupervised learning called the competitive ANN learning model. The model reduces the data dimensions, and it clusters similar data together [25]. A trained SOM network does not require going through the whole DNA sequence to recognize the pathogen but only requires a small part of its DNA. SOM can be highly parallelized, and such parallel implementation has been proposed for synchronous VLSI design [53], custom FPGA [40] and GPUs [32]. Another important aspect of SOM and other NNs is their robustness. NNs work satisfactorily well with low bit resolution without sacrificing much of their accuracy [24]. In this work, we explore the limits of the SOM using different bit resolutions and the effect that bit resolutions have on the accuracy of the SOM, as well as the benefits that this low resolution can provide for a hardware architecture.

## 5.1 Introduction

An emerging design paradigm that can achieve better energy efficiency by trading off the quality (e.g., accuracy) and effort (e.g., energy) of computation is approximate computing [57]. Many modern applications, such as machine learning and signal processing, can produce results with acceptable quality despite most of the calculations being computed imprecisely [54]. The tolerance of imprecise computation in approximate computing to acquire substantial performance gains is the basis for a wide range of architectural innovations [13]. It has been demonstrated that high-precision computations are often unnecessary in the presence of statistical algorithms [33, 56]. Zhang et.al. [56] report less than 5% of quality loss obtained by simulation of the real hardware implemented in a 45nm CMOS technology.

A popular low-power strategy is representing data with reduced bit-widths by trading off accuracy. The benefit of using reduced bit-width is improved energy performance because there is a reduction in the energy cost consumption for data transfers, which usually dominates the total energy consumption for such systems.

Gupta et al. present results where they train deep networks with 16 bits fixed-point number representations and stochastic rounding [19]. Talathi et al. show that the best performance with reduced precision can be achieved with 8 bits weights and 16 bits activation, which, if reduced to 8 bits, results in a 2% drop in accuracy [29]. Hashemi et al. look at a broad range of numerical representations applied to ANNs in both inputs and network parameters and analyze the trade-off between accuracy and hardware implementation metrics, and conclude that a wide range of representations is feasible with negligible degradation in performance [21].

We present a design space exploration of a self-organizing map (SOM) to analyze the impact of different bit resolutions on the accuracy and its benefits. SOM uses a type of unsupervised learning called the competitive ANN learning model. To lower the energy consumption, we exploit the robustness of SOM by successively lowering the resolution to gain efficiency and lower the implementation cost. We do an in-depth analysis of the reduction in resolution vs. loss in accuracy. We present an FPGA implementation of SOM aimed at a bacterial recognition system for battery-operated clinical use where the area, power, and performance are critical. Using this implementation, we demonstrate that a 1% loss in accuracy with 16-bit representation can yield significant savings in energy and area.

## 5.2 Background

In this section, we briefly introduce the SOM algorithm [53], which has been used to recognize bacterial genomes.

As shown in Figure 5.1, a SOM network is organized in a circle to match genomic data better [53]. SOM networks are typically used as 2D arrays when dealing with data that have clear boundaries or start-and-stop conditions. However, for genomic data, especially circular bacterial genomes, it's more accurately represented as a continuous stream without a distinct beginning or end. To prevent any issues from edge effects when presenting data in smaller pieces to the SOM, a circular structure is chosen by Yang et al. [53].

Let us consider a SOM network with $N$ neurons. Each neuron has a weight vector that has the same size as the training vector. Let us assume it to be $M$. Therefore, the weight matrix ($W$) for the SOM network would have $M \times N$ weights. The weight vector of $j^{th}$ neuron is shown as $\left[w_{1,j}, w_{2,j}, \ldots, w_{M,j}\right]$ in Figure 5.1. Each element in the input vectors represents one of the four nucleotides ($C, G, T$or$A$). Let us assume the input sequence ($IS$), is a group of $S$ input vectors ($I_1$, $I_2$, ..., and $I_S$). During training and inference, the distance of the input vectors from each neuron is calculated, shown using arrows in Figure 5.1.

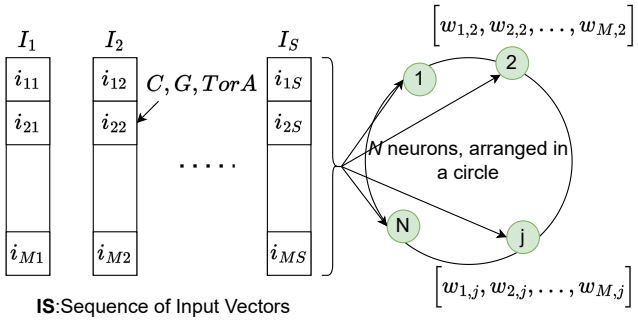



Figure 5.1: SOM based Genomic Identification

The training phase is for a specific bacterial genome. The objective of training a SOM network is to embed genomic features of a specific bacteria into a SOM. Each SOM network can be trained to recognize only one bacteria. In order to recognize a range of bacteria, many SOM networks need to be trained. During inference, when DNA sequences of unknown bacteria are compared with trained SOM networks, the SOM network trained with the same bacteria as the unknown test bacteria must have the highest correlation. Based on the correlation, we can identify the unknown bacteria.

Algorithm 4 describes the actual training and inference processes. During the training process, the algorithm first tries to find the neuron with a minimum distance from the particular input vector. This neuron is considered a winning neuron. Then, the weights of the neighborhood of the winning neuron are updated based on the distance from the winning neuron. The training process is repeated for all training vectors, and the SOM learns the features of that particular bacterial genome.

Random weights initialize the weight matrix. The complete bacterial DNA sequence is chopped randomly into a set of fixed-size training sequences, the $IS$ in Algorithm 4, line 3. On the line 5, parameters $\beta_{min}$, $decay\_factor$, and $\beta$ are initialized. $\beta$ is used to control the converging process of SOM. $\beta$ is decreased by the factor $decay\_factor$ at each step of the converging process, and $\beta_{min}$ is the minimum value $beta$ can attain. In line 7 and 8, the training process first tries finding the winning neuron. Then, weights of the neighborhood of the winning neuron are updated based on the distance between the target neuron and the winning neuron, as shown in line 10 and 11. Finally in line 12, the parameter $\beta$ is decreased by $decay\_factor$. After repeating the training process for all training vectors in the input sequence $IS$, the SOM would have learned the features of that particular bacterial genome.

During the inference phase, some test DNA fragments from one unknown bacteria are sent to each trained SOM network. The SOM network that correlates the test input sequence best is chosen as the winning SOM, and the bacteria it represents reveals the identity of the unknown test bacteria. The correlation measurement is marked by the score shown in Algorithm 4 line 16. The smaller the score is, the better it correlates with the input test vectors.

**Algorithm 4** Pseudo code SOM learning and inference for genome identification

**Algorithm Part 1**: SOM training for 1 bacterial genome

**Input** $N$ : Number of neurons

**Input** $I$=$[i_1, i_2, ..., i_M]$: Input-Vector

**Input** $IS$=$I_1, I_2, ..., I_S$: Sequence of Input-Vectors, each IS represents one bacterial genome

**Input** $W_{i,j} : M \times N$ weight matrix for 1 bacteria

$\beta_{min} = 0.01$; $decay\_factor = 0.99$; $\beta = 1.0$

**for** $I_k \in IS$ **do**

  $dist_{min} = \min_{j=1...N}(\sum_{i=1}^{M} |I_{k,i} - W_{i,j}|)$

  $j_{min} = j$ where $dist_j = dist_{min}$

  **for** $j \in 1...N$ **do**

    $dist = \frac{N}{2} - ||j - j_{min}| - \frac{N}{2}|$ ▷ toroid distance

    $W_j = W_j - \frac{\beta}{2^{dist}}(W_j - I_k)$

  $\beta = \min(\beta * decay\_factor, \beta_{min})$ ▷ decay $\beta$

**Algoritm Part 2**: SOM inference

**Input** $TIS$=$I_1, I_2, ..., I_S$: Test Input Sequence of Input-Vectors for which the bacteria is to be identified

**Input** $W_{r,j,i}$=$R \times N \times M$: Weights for R bacteria

$Inferred\_r$ is $r$ with

$score$= $\min_{r=1...R}\left[\sum_{k=1}^{S} \min_{j=1...N}(\sum_{i=1}^{M} |I_{k,j} - W_{r,i,j}|)\right]$

## 5.3 Low bit-width FPGA Design of SOM

In this section, we present the FPGA implementations that were used to implement SOM for the identification of bacterial genomes. The FPGA implementation is done on Xilinx Virtex7 485t chip to identify bacterial genomes.

A custom semi-systolic array was hand-crafted for each of the different bit width implementations to analyze the area versus energy trade-off. The design takes a vector of weights as input and finds the neuron in the network closest to that input vector. This neuron is referred to as the winning neuron of the network. During the inference phase, the design outputs the distance between the input vector and the closest neuron's weights vector. This distance or score is used for identifying the bacteria in the test samples. During the training phase, neurons' weights are updated according to their distances from the winning neuron

after finding the winning neuron. Figure 5.2 presents a high-level block diagram of the FPGA

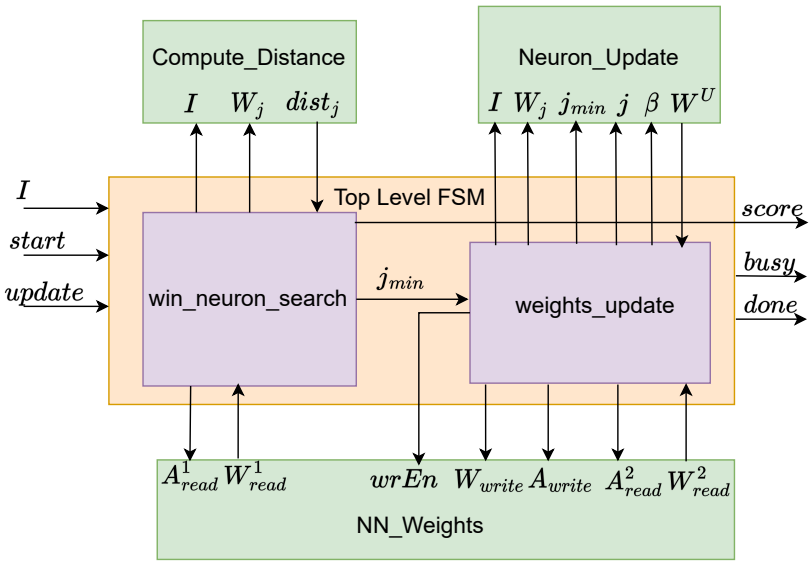



Figure 5.2: High level block diagram of the FPGA Implementation of BioSOM.

implementation of BioSOM. The design takes input as an input vector ($I = [I_1, I_2, ..., I_M]$), where $M$ symbols are represented by pairs of bits corresponding to nucleotides A, C, G, or T. Therefore, the input vector of $M$ symbols is represented by a word of $2M$ bits. The FPGA design serves to infer and train the SOM network, using a $start$ signal to initiate the process and an $update$ signal to distinguish between inference and training operations. The design outputs three signals: $busy$, $score$, and $done$. $busy$ signal is set to 0 when the design is ready to accept the next input vector and 1 when busy processing the current input. $score$ represents the distance of the input vector from the winning neuron as an integer value. The $score$ value is valid after the $done$ signal is set to 1. $done$ signal indicates the completion of the distance computation. The key components of the design are as follows:

- Compute_Distance: This module calculates the distance between the input-vector ($I$) and a single neuron.

- Neuron_Update: This module computes the updated weight vector of a neuron during the training phase.

- NN_Weights: This module reads and updates the Neural Network (NN) weights stored in Block RAM (BRAM).

The top-level FSM (Finite State Machine) iterates through all the neurons in the SOM. For each neuron, it reads the neuron's weights using NN_Weights, computes the distance of the neuron from the input-vector ($I$) using Compute_Distance, and updates the minimum distance ($dist_{min}$) and the index of the winning neuron ($j_{min}$). During the inference phase ($update$=0), the FSM returns $dist_{min}$ as the $score$, determining the matching bacterial genome, and sets the $done$ signal to 1.

During the training phase ($update$=1), after identifying the winning neuron index ($j_{min}$), additional steps are performed to compute and update the weight vectors of neurons. The top-level FSM iteratively reads the neurons' weight vectors using NN_Weights and invokes Neuron_Update. It passes the neuron weights vector ($W$), input vector ($I$), the winning neuron index ($j_{min}$), and network parameter $\beta$ as inputs. Neuron_Update returns the updated neuron weights vector ($W^U$), which is then stored in BRAM by the top-level FSM using NN_Weights.

The Neural Network weights are stored in a 2-dimensional array in BRAMs, where the first dimension represents the neurons in the network, and the second dimension holds the weights of each neuron. Assuming there are N neurons in the network, each neuron has M weights, and each weight is stored as a fixed-point number. The bit-width analysis is performed by varying the number of bits (8, 12, 16, 24, and 32) used to represent the weights. The design is configurable for $N$, $M$, and the number of bits used to represent each weight.

Figure 5.3 illustrates the design of the Compute_Distance module. This module takes an input-vector ($I$) and a neuron weight vector ($W$) as inputs and computes the distance between the neuron weights and the input vector ($\sum_{i=1}^{M} |I_i - W_i|$) using the algorithm described in Algorithm 4.

The Compute_Distance module is implemented with pipelining, and it consists of $M$ pipeline stages. The Initiation Interval (II) of this component is 1, meaning that each stage of

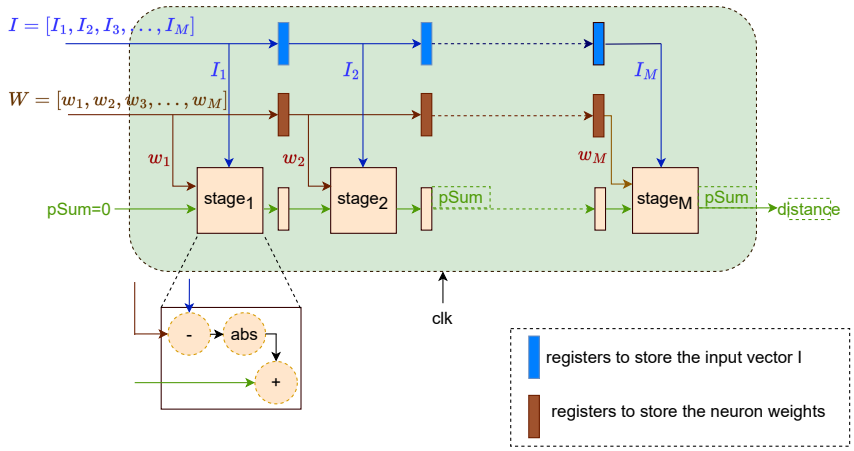



Figure 5.3: Pipelined implementation of Compute_Distance with M stages and Initiation Interval (II)=1 .

the pipeline can be initiated in consecutive clock cycles. Each pipeline stage computes the absolute difference between one element of the input vector and the corresponding neuron weight and accumulates the difference into a partial sum (pSum), as shown in Figure 5.3.

To elaborate, the pSum accumulated in one stage is passed as input to the next stage, creating a data flow through the pipeline. The pSum for the first stage is initialized to zero, and at the last stage, the pSum represents the distance between the input and neuron vectors.

The *Neuron Update* has seven stages pipelined design with II=1. The pipeline stages are shown in Figure 5.4. Neuron_Update takes inputs the weights vector ($W$), index of the neuron ($j$), the input vector ($I$), the wining neuron index ($j_{min}$), and the $\beta$ and outputs the updated weights vector $W_U$. It first computes the distance ($dist$) of the input neuron from the winning neuron (Algorithm 4, line 10) and then computes the update factor ($\frac{\beta}{2^{dist}}$). Neuron_Update then updates the weights using the update factor (Algorithm 4, line 11).

If $N$ is the number of neurons in the SOM network and each neuron has $M$ weights, determining the winning neuron takes $M+N$ cycles. The latency of the inference phase is $M$ and $II$=1. The weight update for an input vector can only start after the top-level FSM determines the winning neuron index ($j_{min}$). After finding the winning neuron, Neuron_Update takes seven cycles to update the weight vector of the first neuron and then

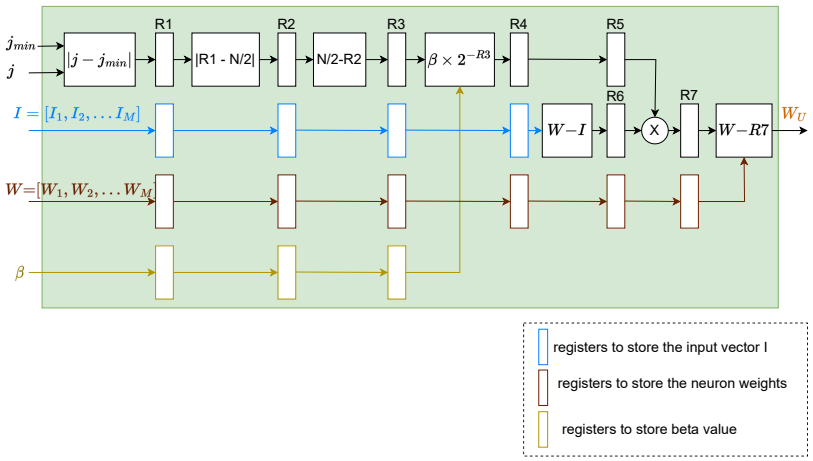



Figure 5.4: Pipelined Implementation of Neuron_Update with 7 stages and Initiation Interval (II)=1.

$N-1$ cycles to update the weights of the remaining $N-1$ neurons. For a given input vector, these two components can execute sequentially. However, while Neuron_Update is processing $i^{th}$ input vector, Compute_Distance can start its operations for the next $(i+1)^{th}$ input vector immediately after the first weight is updated by Neuron_Update at $((M+N)+7)^{th}$ cycle for the $i^{th}$ input vector. In this way Compute_Distance and Neuron_Update components can overlap their execution, processing different input vectors at the same time. This design has overall $II=M+N+7$ cycles for the training phase.

## 5.4 Experimental Results And Analysis

In this section, we present the results of our experiments. In subsection 5.4.1, we give the details of our experimental setup used for measuring the accuracy of the SOM. Subsections 5.4.2 describe the experimental setup and present the results of the FPGA implementation.

### 5.4.1 Accuracy experimental setup

The SOM has been implemented with a range of fixed-point formats. With fewer bits, one naturally expects the SOM network for bacterial identification to suffer from accuracy

degradation. A MATLAB simulation model was created to analyze the accuracy loss using fixed-point implementation. The experiment used a scaled-down version of the bacterial identification problem to reduce the training time of the network. We consider that this does not compromise the resulting accuracy of the SOM. We trained 10 SOMs with ten different bacteria DNA sequences. Each SOM network has 100 neurons inside, and each neuron has 20 weights. We trained the networks with two independent training processes running in parallel. One is implemented using double precision floating point, and the other is implemented with fixed-point weights. After training, we used the trained networks to identify the unknown sequence and record their scores.

Two metrics are defined to evaluate the implementation accuracy. The *quantization error* is defined as the relative error between fixed-point format score ($S_{fixed}$) and double-precision floating-point format score ($S_{float}$):

$$quantization\_error = \frac{|S_{fixed} - S_{float}|}{S_{float}} \times 100\%$$

Assuming the double precision floating format has no classification error, the *classification error* for the fixed-point format is defined as the ratio of the number of times the network classified ($C_{false}$) the bacterial strain to the total number of tests performed ($C_{total}$):

$$classification\_error = \frac{C_{false}}{C_{total}} \times 100\%$$

Classification and quantization errors are not entirely independent but reveal the different aspects of fixed-point implementation accuracy. Quantization error shows the difference between fixed-point representation and golden double-point reference. It does not change with the problem. Classification error determines the final impact of the accuracy loss introduced by the fixed-point implementation and is problem dependent. Figure 5.5 presents the quantization and classification errors depending on the bit resolution. From the figure, 8-bit

representation completely fails the experiment, 12-bit has 39% classification error, 16-bit has <1%, and 24-bit and 32-bit successfully pass the experiment with 0% of classification error. However, we should notice that though 16-bit pass the test, it still has about 10% quantization error.

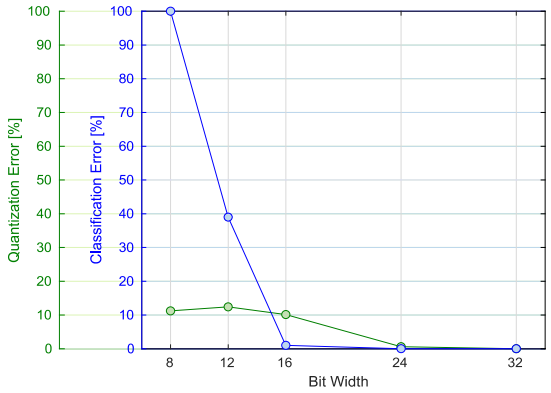



Figure 5.5: Quantization error compared to the identification error.

Each SOM was trained with a specific fixed-point resolution. The quantization error shows the difference of the *score* variable, see algorithm Algorithm 4, between the fixed-point and the double precision. The quantization error is very important for the *score* variable of the SOM, since this variable is the one that decides which is the winning SOM, i.e., which SOM matches the input DNA sequence. It is essential to guarantee sufficient bits for the *score* of each SOM. If the resolution is too low, the *score* of the SOMs after quantization will be identical even with low quantization error, making it impossible to identify the bacteria correctly. This explains that the quantization error is low in Figure 5.5, but the classification error is high in the 8 to 12-bit resolution region.

### 5.4.2 FPGA experimental setup

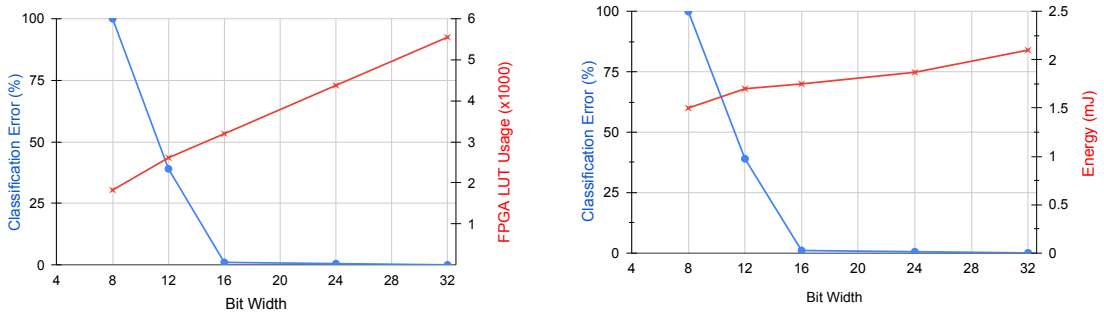
This section presents the experimental results of the FPGA implementations presented in section 5.3. The FPGA design is implemented with Vivado v.2016.4 used to synthesize and analyze the HDL Designs. Our design is implemented in VHDL and validated using the

Vivado simulator. Experimentation is done for different fixed point representations of weights by modifying parameters in VHDL code.

Table 5.1: Resource Comparison of different fixed point formats

| Resource | 8b | 12b | 16b | 24b | 32b |
|---|---|---|---|---|---|
| LUTs | 1823 | 2611 | 3196 | 4375 | 5549 |
| Registers | 3481 | 4679 | 5871 | 8255 | 10639 |
| Slice | 854 | 1158 | 1369 | 1809 | 2395 |
| LUT FF Pairs | 1007 | 1369 | 1750 | 2372 | 3043 |
| B-RAM | 4 | 6 | 8 | 11 | 15 |
| DSP48E1 | 17 | 17 | 17 | 17 | 33 |
| Bonded IOB | 57 | 61 | 65 | 73 | 81 |
| Power(W) | 8b | 12b | 16b | 24b | 32b |
| Total Power | 0.295 | 0.314 | 0.332 | 0.356 | 0.392 |
| Dynamic | 0.052 | 0.071 | 0.089 | 0.113 | 0.148 |
| Device Static | 0.243 | 0.243 | 0.243 | 0.244 | 0.244 |

The area and power numbers for different weight resolutions are extracted from the reports generated by the Vivado tool post placement and routing with a working frequency of 100 MHz. Table 5.1 compares the resources and area for 8, 12, 16, 24, and 32 bits fixed point formats for a SOM network with 512 neurons. The second part of the table compares the average power in the different fixed point formats for the same SOM.




(a) FPGA LUT utilization (b) energy of FPGA implementation

Figure 5.6: Area and energy comparison for different fixed-point format.

The results are summarized in Figure 5.6a and Figure 5.6b. Both the amount of utilized

LUTs and total energy in Joule is presented against the classification error. From the figures, we can easily conclude that we can substantially reduce the resources used and the energy by using a 16-bit fixed-point representation without losing accuracy. We can reduce the resources even further by moving to the 12-bit representation by sacrificing 39% the SOM accuracy.

## 5.5 Summary

This chapter discusses using Self-Organizing Maps (SOM) for rapid genome identification, specifically for bacterial genomes. The chapter explores the benefits and limitations of using low bit-width resolution for representing weights in the SOM and its impact on accuracy, resource utilization, and energy efficiency in hardware implementations.

We present a design space exploration of SOM using different bit resolutions (8, 12, 16, 24, and 32 bits) for representing the neural network weights. We analyze the impact of these bit resolutions on the accuracy of SOM and their benefits for hardware implementation. The trade-off between accuracy and hardware efficiency is explored by training multiple SOMs with different bit representations and comparing their performance.

The experimental setup includes a MATLAB simulation model to measure accuracy loss when using fixed-point implementations of the SOM. The quantization and classification errors are metrics to evaluate the accuracy of different bit resolutions. The results show that an 8-bit representation completely fails the experiment, while 12-bit has a high classification error but a lower quantization error. 16-bit representation exhibits minimal classification error but still has a quantization error of around 10%. On the other hand, 24-bit and 32-bit representations successfully pass the experiment with 0% classification error.

The FPGA implementation of SOM is presented, aiming at bacterial genome identification. The design is configurable for different numbers of neurons, weight dimensions, and bit resolutions. We utilize pipelining and parallelization techniques to optimize the hardware design for efficiency. Resource utilization and power consumption are

compared for different bit representations.

The experimental results show that a 16-bit fixed-point representation provides a good balance between resource utilization, energy efficiency, and accuracy. It significantly reduces resources compared to higher-bit representations (24 and 32 bits) while maintaining almost the same accuracy. A 12-bit representation further reduces resource usage but sacrifices accuracy.

In conclusion, the chapter highlights the benefits of using low bit-width resolution in the hardware implementation of SOM for genome identification. A 16-bit representation is found to be a practical choice, offering significant savings in resources and energy consumption without compromising accuracy. The study provides valuable insights into the trade-offs in optimizing SOM for efficient hardware implementations in genome recognition applications.

# Chapter 6

# Conclusion And Future Directions

## 6.1 Introduction

The past few years have witnessed remarkable growth in Neural Network (NN) based applications, with significant contributions in healthcare, agriculture, road safety, surveillance, defense, and many others. The availability of modern computing systems capable of handling large datasets and the development of deep learning frameworks like TensorFlow and PyTorch have propelled NNs to achieve human-like performance. These advancements have led to the adoption of NNs in various domains, including embedded systems with limited computing resources and energy budget constraints.

The objective of this thesis was to improve the energy efficiency and throughput of NN accelerators on parallel architectures, considering the challenges posed by limited on-chip memory and high off-chip memory accesses. To achieve this goal, we focused on data reuse techniques, data partitioning, and scheduling strategies for different types of NNs.

## 6.2 Recapitulation of Research Objectives

This research aimed to enhance the energy efficiency and throughput of Neural Network (NN) accelerators by minimizing off-chip memory accesses. In Chapter 1, we defined our research objectives to address the challenges of energy efficiency and throughput in NN accelerators. This thesis explored various data reuse techniques for different types of NNs, including Self Organizing Maps (SOMs), Convolutional Neural Networks (CNNs), and Recurrent Neural Networks (RNNs). An analytical framework was also developed to quantify off-chip memory accesses and compare different data partitioning and scheduling schemes, considering the architectural constraints, for optimizing NN performance on parallel architectures.

## 6.3 Summary of Key Findings

Chapter 2 presented an analytical framework that quantifies off-chip memory accesses for DNN layers with varying shapes, considering architectural constraints. This framework allowed us to compare different data partitioning and scheduling schemes and identify the optimal solutions for improving energy and throughput.

In Chapter 3, we proposed a data reuse approach for CNN layers that considers architectural parameters and determines the optimal partitioning and scheduling scheme. Our proposed approach significantly improved energy efficiency and throughput for memory-intensive CNN layers.

Chapter 4 addressed the challenges of data reuse in recurrent neural networks (RNNs), particularly in Long-Short Term Memory Networks (LSTMs). We proposed a novel data reuse approach that efficiently reuses weights for consecutive time steps, significantly reducing off-chip memory accesses and improving energy efficiency.

Chapter 5 explored the impact of different bit resolutions on NN accuracy, focusing on Self-Organizing Maps (SOMs). We implemented different bit-width configurations on an FPGA and

compared their performance, providing insights into energy-constrained systems' design.

## 6.4 Implications and Future Research Directions

The findings of this thesis have implications for optimizing NN accelerators in a wide range of applications, particularly in edge AI devices with limited energy and computing resources. The data reuse techniques and partitioning strategies can be extended and applied to modern AI models, such as Language Models (LLMs) and Stable Diffusion networks, to improve their energy efficiency and throughput.

Future research directions could involve exploring hardware co-design and architectural optimizations tailored specifically for NN accelerators. Additionally, investigating more advanced quantization and pruning techniques to strike a better balance between accuracy and energy efficiency would be valuable.

We also suggest exploring the incorporation of hardware accelerators specialized for specific NN tasks to improve energy efficiency and performance further. Such accelerators can exploit the specific data access patterns and computations inherent in NN layers.

Moreover, addressing the challenges posed by data reuse in more complex and deeply layered neural networks would be an interesting area of research. Investigating techniques to optimize data reuse in multi-layer and multi-modal NNs could lead to more efficient and powerful edge AI devices.

Future research in genome identification using Self-Organizing Maps (SOMs) could benefit from two key strategies. First, exploring a co-training approach—simultaneously training multiple SOM models on the same dataset—may enhance the model's ability to learn complex genomic patterns. Second, considering weight initialization, using techniques like Principal Component Analysis (PCA) could offer a more informed start for the model, potentially improving its accuracy. These future directions aim to refine SOM training, leading to more accurate and efficient genome identification.

## 6.5 Closing Remarks

In conclusion, this thesis has contributed to the state-of-the-art in energy-efficient execution of modern NNs on parallel architectures. The proposed data reuse approaches and partitioning strategies have significantly improved energy efficiency and throughput for various NN layers. The analytical framework developed in this research provides a valuable tool for comparing different data reuse and scheduling schemes and exploring the design space for optimal solutions.

The work presented here not only advances the field of NN accelerators but also opens up new research directions for integrating modern AI models into energy-constrained systems. By focusing on efficient data reuse, hardware co-design, and innovative quantization techniques, future researchers can continue to push the boundaries of NN performance on parallel architectures.

## 6.6 Conclusion

This thesis has successfully addressed the challenges of optimizing neural network performance on parallel architectures with limited on-chip memory. Through the development of novel data reuse approaches, data partitioning, and scheduling strategies, we have significantly improved energy efficiency and throughput for various NN layers.

The findings of this research have broader implications for the design of efficient edge AI devices and contribute to the ongoing progress in the field of neural network accelerators. The proposed techniques can be extended and applied to modern AI models, such as Language Models (LLMs) and Stable Diffusion networks, to further enhance their performance on parallel architectures.

As the field of artificial intelligence continues to evolve, optimizing neural network performance will remain a critical area of research. We hope that this thesis serves as a

stepping stone for future researchers to explore innovative solutions and further advance the efficiency and capabilities of NN accelerators.

With this, we conclude the thesis, and we are confident that the knowledge gained from this research will contribute to the continued advancement and practical implementation of energy-efficient neural network accelerators on parallel architectures.

# Bibliography


[1] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. Compilers: Principles, techniques, and tools, 2006.

[2] M. Alwani, H. Chen, M. Ferdman, and P. Milder. Fused-layer CNN accelerators. In *MICRO*, 2016.

[3] ARM. *AMBA AXI Protocol Specification [Online]. Available at https://developer.arm.com/docs*, 2010. Rev. 2.0.

[4] E. Azari and S. Vrudhula. Elsa: A throughput-optimized design of an lstm accelerator for energy-constrained devices. *ACM Transactions on Embedded Computing Systems (TECS)*, 19(1):1–21, 2020.

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

[6] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam. DianNao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. *ASPLOS*, 2014.

[7] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, et al. DaDianNao: A machine-learning supercomputer. In *MICRO*, 2014.

[8] Y.-H. Chen, J. S. Emer, and V. Sze. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. *ACM/IEEE ISCA*, 2016.

[9] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE Journal of Solid-State Circuits*, 2016.

[10] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer. cuDNN:efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759*, 2014.

[11] F. Conti, L. Cavigelli, G. Paulin, I. Susmelj, and L. Benini. Chipmunk: A systolically scalable 0.9 mm 2, 3.08 gop/s/mw@ 1.2 mw accelerator for near-sensor recurrent neural network inference. In *2018 IEEE Custom Integrated Circuits Conference (CICC)*, pages 1–4. IEEE, 2018.

[12] Z. Du, R. Fasthuber, T. Chen, P. Ienne, L. Li, T. Luo, X. Feng, Y. Chen, and O. Temam. ShiDianNao: Shifting vision processing closer to the sensor. In *ISCA*, 2015.

[13] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger. Neural acceleration for general-purpose approximate programs. In *45th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 449–460, 2012.

[14] J. C. Ferreira and J. Fonseca. An fpga implementation of a long short-term memory neural network. In *2016 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–8. IEEE, 2016.

[15] J. S. Garofolo. Timit acoustic phonetic continuous speech corpus. *Linguistic Data Consortium, 1993*, 1993.

[16] V. Gokhale, J. Jin, A. Dundar, B. Martini, and E. Culurciello. A 240 G-ops/s mobile coprocessor for deep neural networks. In *CVPR Workshops*, 2014.

[17] Y. Guan, Z. Yuan, G. Sun, and J. Cong. Fpga-based accelerator for long short-term

memory recurrent neural networks. In *2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 629–634. IEEE, 2017.

[18] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan. Deep learning with limited numerical precision. In *ICML*, 2015.

[19] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan. Deep learning with limited numerical precision. In *Proceedings of the 32Nd International Conference on International Conference on Machine Learning*, pages 1737–1746, 2015.

[20] 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 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 75–84, 2017.

[21] S. Hashemi, N. Anthony, H. Tann, R. I. Bahar, and S. Reda. Understanding the impact of precision quantization on the accuracy and energy of neural networks. In *Proceedings of the Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1478–1483, 2017.

[22] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *CVPR*, 2016.

[23] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[24] L. Jiao, C. Luo, W. Cao, X. Zhou, and L. Wang. Accelerating low bit-width convolutional neural networks with embedded fpga. In *27th International Conference on Field Programmable Logic and Applications*, pages 1–4, Sep. 2017.

[25] T. Kohonen. Essentials of the self-organizing map. *Neural Netw.*, 37:52–65, Jan. 2013.

[26] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *NIPS*, 2012.

[27] M. Lee, K. Hwang, J. Park, S. Choi, S. Shin, and W. Sung. Fpga-based low-power speech recognition with recurrent neural networks. In *2016 IEEE International Workshop on Signal Processing Systems (SiPS)*, pages 230–235. IEEE, 2016.

[28] J. Li, G. Yan, W. Lu, S. Jiang, S. Gong, J. Wu, and X. Li. SmartShuttle: Optimizing off-chip memory accesses for deep learning accelerators. *DATE*, 2018.

[29] D. D. Lin and S. S. Talathi. Overcoming challenges in fixed point training of deep convolutional networks. *arXiv preprint arXiv:1607.02241*, 2016.

[30] Y. Ma, Y. Cao, S. Vrudhula, and J.-s. Seo. Optimizing loop operation and dataflow in fpga acceleration of deep convolutional neural networks. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '17, page 45–54, 2017.

[31] K. T. Malladi, F. A. Nothaft, K. Periyathambi, B. C. Lee, C. Kozyrakis, and M. Horowitz. Towards energy-proportional datacenter memory with mobile dram. In *ISCA*, 2012.

[32] S. McConnell, R. Sturgeon, G. Henry, A. Mayne, and R. Hurley. Scalability of self-organizing maps on a GPU cluster using OpenCL and CUDA. *Journal of Physics: Conference Series*, 341:012018, feb 2012.

[33] B. Moons, R. Uytterhoeven, W. Dehaene, and M. Verhelst. 14.5 envision: A 0.26-to-10tops/w subword-parallel dynamic-voltage-accuracy-frequency-scalable convolutional neural network processor in 28nm fdsoi. In *IEEE International Solid-State Circuits Conference (ISSCC)*, pages 246–247, 2017.

[34] M. Motamedi, P. Gysel, V. Akella, and S. Ghiasi. Design space exploration of fpga-

based deep convolutional neural networks. In *2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 575–580, 2016.

[35] J. Park, J. Kung, W. Yi, and J.-J. Kim. Maximizing system performance by balancing computation loads in lstm accelerators. In *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 7–12. IEEE, 2018.

[36] J. Park, W. Yi, D. Ahn, J. Kung, and J.-J. Kim. Balancing computation loads and optimizing input vector loading in lstm accelerators. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(9):1889–1901, 2019.

[37] 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*, pages 13–18, 2020.

[38] M. Peemen, B. Mesman, and H. Corporaal. Inter-tile reuse optimization applied to bandwidth constrained embedded accelerators. In *2015 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 169–174, 2015.

[39] M. Peemen, A. A. A. Setio, B. Mesman, and H. Corporaal. Memory-centric accelerator design for convolutional neural networks. In *2013 IEEE 31st International Conference on Computer Design (ICCD)*, pages 13–19, 2013.

[40] M. Porrmann, U. Witkowski, and U. Rückert. *Implementation of Self-Organizing Feature Maps in Reconfigurable Hardware*, pages 247–269. Springer US, Boston, MA, 2006.

[41] 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)*, volume 2160, pages 17–24. IEEE, 2019.

[42] V. Rybalkin, A. Pappalardo, M. M. Ghaffar, G. Gambardella, N. Wehn, and M. Blott. Finn-l: Library extensions and design trade-off analysis for variable precision lstm

networks on fpgas. In *2018 28th international conference on field programmable logic and applications (FPL)*, pages 89–897. IEEE, 2018.

[43] R. Shi, Z. Xu, Z. Sun, M. Peemen, A. Li, H. Corporaal, and D. Wu. A locality aware convolutional neural networks accelerator. In *2015 Euromicro Conference on Digital System Design*, pages 591–598, 2015.

[44] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[45] M. Sundermeyer, H. Ney, and R. Schlüter. From feedforward to recurrent lstm neural networks for language modeling. *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, 23(3):517–529, 2015.

[46] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer. Efficient processing of deep neural networks: A tutorial and survey. *Proceedings of the IEEE*, 105(12):2295–2329, 2017.

[47] F. Tu, S. Yin, P. Ouyang, S. Tang, L. Liu, and S. Wei. Deep convolutional neural network architecture with reconfigurable computation patterns. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2017.

[48] 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*, pages 11–20, 2018.

[49] Z. Wang, J. Lin, and Z. Wang. Accelerating recurrent neural networks: A memory-efficient approach. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 25(10):2763–2775, 2017.

[50] X. Wei, Y. Liang, and J. Cong. Overcoming data transfer bottlenecks in FPGA-based DNN accelerators via layer conscious memory management. In *DAC*, 2019.

[51] L. Xie, X. Fan, W. Cao, and L. Wang. High throughput CNN accelerator design based on FPGA. In *FPT*, 2018.

[52] Xilinx. *AXI Performance Monitor v5.0 [Online]. Available at https://www.xilinx.com/support/documentation/*, 2017. v5.0.

[53] Y. Yang, D. Stathis, P. Sharma, K. Paul, A. Hemani, M. Grabherr, and R. Ahmad. RiBoSOM. In *Proceedings of the 18th International Conference on Embedded Computer Systems Architectures, Modeling, and Simulation - SAMOS*, pages 105–114, New York, New York, USA, 2018. ACM Press.

[54] R. Ye, T. Wang, F. Yuan, R. Kumar, and Q. Xu. On reconfiguration-oriented approximate adder design and its application. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 48–54, 2013.

[55] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong. Optimizing FPGA-based accelerator design for deep convolutional neural networks. In *FPGA*, 2015.

[56] Q. Zhang, T. Wang, Y. Tian, F. Yuan, and Q. Xu. Approxann: An approximate computing framework for artificial neural network. In *Proceedings of the Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 701–706, San Jose, CA, USA, 2015. EDA Consortium.

[57] Q. Zhang, F. Yuan, R. Ye, and Q. Xu. Approxit: An approximate computing framework for iterative methods. In *Proceedings of the 51st Annual Design Automation Conference*, pages 97:1–97:6, 2014.

# List of Publications

This thesis is based on the following publications:

1. **S. Tewari**, A. Kumar and K. Paul, "*SACC: Split and Combine Approach to Reduce the Off-chip Memory Accesses of LSTM Accelerators*", in DATE 2021.

2. **S. Tewari**, A. Kumar and K. Paul, "*Minimizing Off-Chip Memory Access for CNN Accelerators*", in IEEE Consumer Electronics Magazine 2021.

3. **S. Tewari**, A. Kumar and K. Paul, "*Bus Width Aware Off-Chip Memory Access Minimization for CNN Accelerators*", in ISVLSI 2020.

4. D. Stathis, Y. Yang, **S. Tewari**, A. Hemani, K. Paul, M. Grabherr and R. Ahmad, "*Approximate Computing Applied to Bacterial Genome Identification using Self-Organizing Maps*", in ISVLSI 2019.

# Biography

Saurabh Tewari is a Ph.D. student at the Department of Computer Science and Engineering at IIT Delhi. He also works as a senior staff engineer with Qualcomm India. Saurabh holds a Bachelor's degree (B.Tech.) from the Indian Institute of Technology (IIT) Roorkee and completed his Master's degree at IIT Delhi in 2013.

At IIT Delhi, Saurabh has been recognized for his teaching abilities and awarded as an Outstanding Teaching Assistant (TA). His research interests encompass mapping algorithms on parallel and reconfigurable architectures, machine learning, deep neural networks, software compilers, and computer architectures.