OPTIMIZING NEURAL NETWORKS PERFORMANCE ON PARALLEL ARCHITECTURES

SAURABH TEWARI

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY DELHI
NOVEMBER 2022

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

NOVEMBER 2022

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.

Certain works included in this thesis involved collaboration or use of measurement data obtained by other students, which have been explicitly specified/acknowledged in the corresponding chapters and the part done by those collaborators appeared in their respective reports/theses.

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

Abstract

Recent past has observed explosive growth in number of Neural Network (NN) based applications. Their ability to solve long outstanding problems with high accuracy have enabled new set of applications for consumer electronics and embedded devices. Their high accuracy comes at the cost of large number of compute and memory intensive operations. Few years back, NN applications were mostly run on the cloud are now prefferred to run on the edge device to mitigate connectivity, network bandwidth, latency and privacy issues. Several latest consumer electronic devices like smart phones, digital camera etc., are already equipped with NN applications. However, limited compute and energy resource on these edge devices pose a significant challenge. Researchers have proposed several custom hardware accelerators to address these issues. Throughput and energy consumption are the key performance metrics for these edge NN accelerators. These accelerators can meet the performance requirements upto some extend but their energy consumption is still concerning. Efficient processing of the NNs is of paramount importance for their widespread usage on edge devices. A significant fraction of the energy consumption of these accelerators results from accessing the data from the off-chip memory accesses. Their performance is also limited by the memory bandwidth. Optimizing the off-chip memory accesses is the key to improve the performance and energy efficiency of these accelerators. This thesis focuses on efficient inferencing of NNs on edge devices by optimizing the off-chip memory accesses at pre-inferencing step.

Computations in NN layers involve accessing multi-dimensional data from off-chip memory. Different NNs and different layers within a NN exhibit different kind of data access pattern. Even, data access pattern varies among the same type of layers within a NN due to varying layer's shape and sizes. There is no global optimal scheme which works for all. Off-chip memory accesses need to be analyzed for each case to find the optimal solution. Extracting the off-chip memory accesses directly from the NN description is difficult as it depends not only on the layer shape but also on the layer partitioning, scheduling and accelerator's architectural parameters. This thesis proposes an analytical framework that computes the off-chip memory accesses for different types of NN layers and estimates the data

movement energy for off-chip memory accesses. The analytical framework is used to compare various design choices and trade-off between them. It guides for finding an energy efficient solution for different NNs.

There are different type of NNs which are specialized for solving specific problems. These NNs differ each other in terms of number of layers, number of parameters, data flow, training method and the type of operations. In this thesis we have covered range of NNs varying from single layer, multi-layer feedforward NNs to recurrent NNs. The data access pattern and type of data reuse varies among these NNs. Self Organizing Maps (SOM) is an example of single layer feedforward NN which is an unsupervised learning algorithm, used in dimensionality reduction and clustering related problems. We have implemented a custom pipelined FPGA design to accelerate the SOM processing. The target application is to identify bacterial genome in a clinical environment settings where optimizing energy, area is critical. Using the FPGA design, we have analyzed the impact of varying data resolution on the accuracy of the network and energy and area benefits.

Convolution Neural Networks (CNNs) are multi layer feedforward neural networks and have shown tremendous success in computer vision related applications like object detection, image classification, face detection etc. CNNs are quintessential example of deep neural networks (DNNs). CNNs primary use a mathematical operation called convolution. There are several convolution layers (CL) and few fully connected layers (FCL) at the end, which are special case of CL. Other layer types like pooling and normalization are also present in CNNs. However processing of CLs dominate the overall computations and energy consumption of CNNs. NN accelerators have small on-chip memory and CNN layer data size doesn't fit in the memory. Loop tiling is applied to partition the layer data into tiles. There are different ways in which layer data can be partitioned and the order in which operation on these tiles can be performed. Finding the optimal way is not trivial here. To address this we have developed a model to compute the memory access of CLs and FCLs and integrated it with the analytical framework to analyze the memory accesses of different schemes. We have proposed a scheme to determine the optimal layer partitioning and scheduling scheme that maximizes the data reuse while considering the accelerator's architecture parameters, address alignment and, data resolution.

Recurrent Neural Networks (RNN) are other important category of NNs widely used for sequential data processing in speech recognition, natural language processing and other areas. They have recurrent connections and have internal states to store the information from the past. Long Short-Term Memory (LSTM) are variants of RNNs, designed to handle long-range dependencies. These networks access the weight matrices repeatedly for large number of time

steps. Due to dependency on previous step computations, LSTMs accelerators fails to reuse the data and incur large volume of data accesses, resulting in high energy consumption. Throughput of LSTM accelerators is also limited by memory bandwidth. In this work, we have proposed a data reuse scheme to overcome the data-dependency problem of LSTMs that significantly improve the data reuse and improves the throughput and energy consumption of these accelerators.

Overall, this thesis proposes an analytical framework to compute the memory accesses and analyze the data movement energy of state of the art NNs and proposes novel data-reuse schemes to optimize the off-chip memory accesses. This thesis contributes the state-of-the-art by improving the energy efficiency and throughput of NN accelerators during inference phase.

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 Background

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.

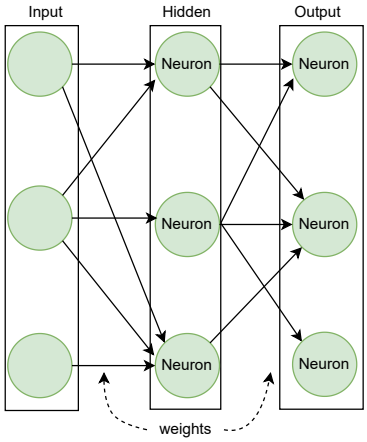


Figure 1.1: Example of simple Neural Network.

There are several classes of NNs that differ from each other in the number of neurons, connections between them, and method of training. 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, while LSTMs are used in sequential data processing applications like speech recognition and natural language processing (NLP).

Modern DNN architectures are very deep and involve many layers and millions of parameters. Multilayer NNs with more than three layers are referred to as deep neural networks (DNNs). These DNNs are capable of learning complex functions from the data. Besides DNNs, there are single-layer NNs that consist of just two layers - an input and an output layer. Only the output layer performs the computation in this case. Self-Organizing Maps (SOMs) are examples of single-layer NNs and are used in dimensionality reduction and clustering applications.

1.1.1 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 go through a learning process 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 go

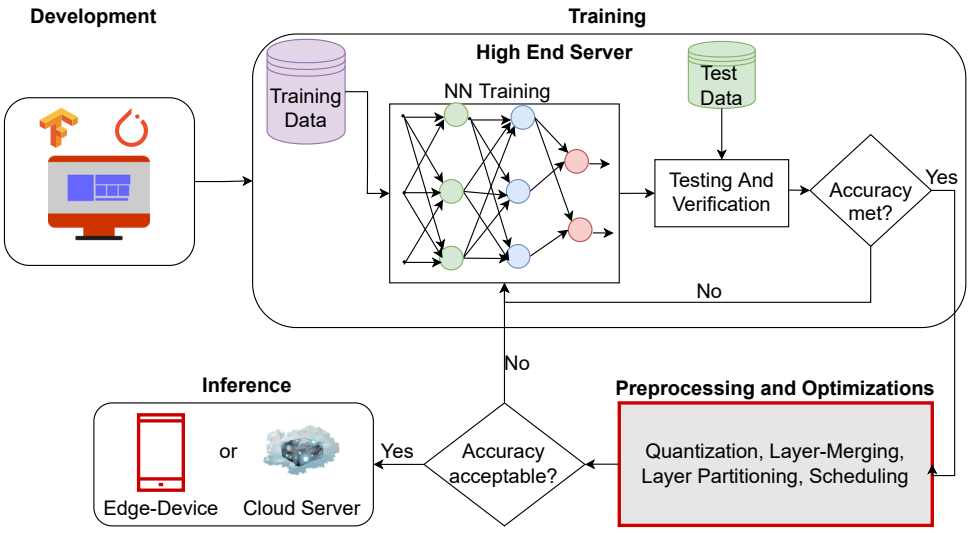


Figure 1.2: Work Flow for Neural Networks.

through training phase where they learn from the real examples. For learning, these DNNs require large training data sets and must go through multiple iterations to achieve the desired accuracy. Training of NNs involves updating weights and biases using large input samples. Its a repetitive process, until the desired accuracy is achieved. Training a DNN may last from few hours to several weeks. Hence training of DNNs are 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. A popular CNN, VGG16 performs 15.47 GMACs 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 the edge devices is a challenge due to limited energy and compute resources. Hence optimizing these models before inferencing on edge devices is a must. In this work, we focused on optimizing NN models before their deployment on edge-devices for inferencing. This phase is shown as **Preprocessing and Optimization** block in Figure 1.2.

1.2 Edge AI: Inferencing on Edge Devices

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

Recent growth of deep learning, advancement in deep learning tools, and recent research in the field of efficient edge AI accelerators have enabled several intelligent applications for consumer and edge devices. Today we see explosion of applications using deep learning algorithms in consumer electronic devices. Manufacturers are shifting the processing of NNs from cloud to edge devices like smartphones and tablets.

However DNN inferencing on the 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.

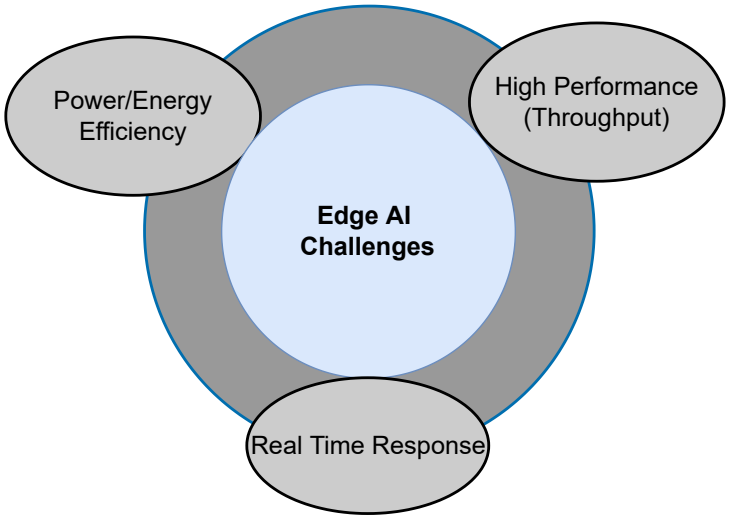


Figure 1.3: Key challenges for Edge-AI Accelerator

1.2.1 NN Accelerators

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. 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 different storage capacity, access latency and energy costs. The memory access energy from off-chip memory is up to two orders of magnitude higher than a PE computation operation [8]. It has been observed that more than 80% of the overall energy consumption of these accelerators is 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. Most recent research has focused on reducing off-chip memory accesses.

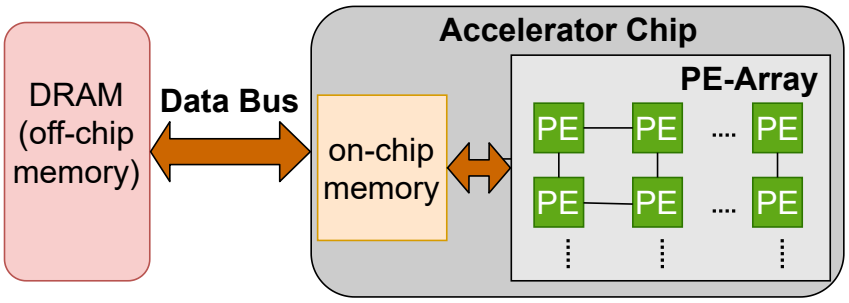


Figure 1.4: Typical DNN accelerator architecture.

In this thesis, we desired to improve edge NN accelerators' performance and energy efficiency during the inference phase. We aimed to provide energy-efficient solutions for NN accelerators by reducing the off-chip memory accesses by applying novel data reuse schemes and representing data in low resolution.

1.3 Previous Work

Several FPGA [2, 16, 18, 43, 45, 49], 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. Since memory accesses dominate the performance and energy of these accelerators, recent researches focus on reducing the off-chip memory accesses [6, 9, 49]. Some approaches [14, 27, 37] used on-chip memory to store all the weights. Sizes of weights in modern NN models can be several MBs. These approaches are not scalable and effective only for small NN models. Works that aim 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

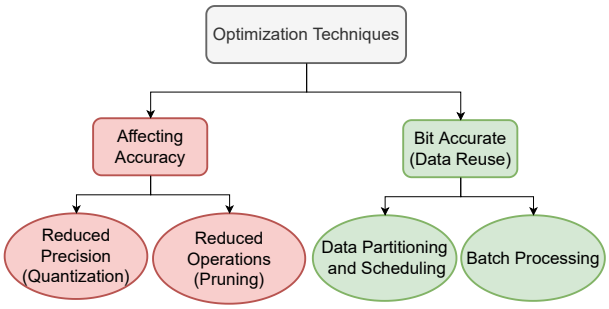


Figure 1.5: Broad Classification of previous works for improving the performance of DNN accelerators.

models' size [5, 14, 20, 27, 41]. With reduced precision, the storage requirement and memory accesses reduce proportionally and improve energy efficiency [40]. 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. Also, the number of parameters in modern DNNs is significantly large. For these DNNs, besides quantization and pruning, additional techniques are required to reduce the off-chip memory accesses further.

Data-reuse schemes are the other line of approach that does not affect the accuracy of the network. 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 [28, 34, 36, 49]. One popular scheme which reuses weights is batch processing [36]. 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 not desirable. Secondly, the batch processing scheme only improves the weights reuse, and other layer data types do not get the benefit.

Another prominent data reuse technique is data partitioning and scheduling [49, 49]. 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. For a given DNN, there are numerous ways of data partitioning and scheduling, offering different extents of data reuse. Loop tiling is applied to partition the layer data, a well-known compiler technique. Conventional problems apply loop tiling where equal data-reuse opportunities exist in all the data dimensions. However, in NN 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 essential 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. Choosing an optimal way here is non-trivial.

We have explored both approaches in this work and contributed some new ideas. We have applied quantization techniques on Self Organizing Maps (SOMs), a single layer FFNN, to analyze the impact on NN accuracy and benefits on improving energy efficiency. We hand-crafted a custom semi-systolic array design for different bit width implementations to analyze the accuracy versus energy trade-off for SOMs. We also proposed novel data reuse approaches for multilayer feedforward and recurrent NNs.

To determine the optimal scheme, estimating the off-chip memory accesses is essential. From the description of the NN layer, it is difficult to estimate the off-chip memory accesses. Several factors impact the off-chip memory accesses of the NN layer. 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.

We have analyzed the benefits of different partitioning and scheduling schemes for popular CNNs. There are several possible ways to partition the layer data and schedule the operations. Finding the optimal partitioning and scheduling scheme by performing the measurements on the hardware is time-consuming, and vast search space makes it practically impossible. We expressed determining the optimal solution that minimizes off-chip memory accesses of a NN layer as a constraint optimization problem. We have developed a model and integrated it with the analytical framework for computing the off-chip memory accesses of CNN layers. We have analyzed the impact of the accelerator's architectural parameters, e.g., bus widths, address alignments, and on-chip memory sizes on off-chip memory accesses of CNN layers.

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.

RNNs computations involve repeated access to the same weight matrices for long sequences. Dependency on previous time step computations and limited on-chip memory of accelerators results in large off-chip memory accesses. To overcome this, we proposed a novel data reuse approach for RNNs that significantly reduces the off-chip memory accesses by

reusing the weights for two consecutive time steps. A key characteristic of the proposed approach is that it 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.

1.4 Summary and Outline of the Thesis

Recent advancements in NNs have enabled them to solve several real-world problems, which researchers long ago considered difficult. Due to their high accuracy, they are now being used in many domains. The last few years have witnessed enormous growth in the number of intelligent applications targeted for edge devices. However, resource and energy-constrained edge devices pose a significant challenge to the efficient processing of NN applications. Hence there is a pressing need for energy-efficient execution of NNs applications for their wide acceptance. In the NN inference phase, a large amount of energy consumption results from expensive off-chip memory accesses. Optimizing off-chip memory accesses is key to improving these edge devices' performance and energy efficiency.

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, taking into account 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

Analytical Framework

NN accelerators use on-chip memory and reuse the data from it to minimize off-chip memory accesses. Figure 2.1 illustrates the impact of on-chip memory data reuse on data access energy. Figure 2.1a shows the memory accesses and energy estimates when N bytes are directly accessed from the off-chip memory and Figure 2.1b shows the reduction in off-chip memory accesses when re-using the data from on-chip memory. If the energy per bytes access from the off-chip and on-chip memories are e_1 and e_2, respectively, the energy efficiency can be expressed as,

$$E_{efficiency} = 1 - \frac{E_2}{E_1} = 1 - (\frac{1}{n} + \frac{e_2}{e_1}) \tag{2.1}$$

For a given architecture, e_1 and e_2 are fixed and e_1 is significantly high compared to e_2. Equation 2.1 shows energy efficiency mainly depends on data reuse and improves significantly with data reuse. 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.

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 flow of the data is shown in Figure 2.2. Due to large difference between the energy consumption of accessing the data from the off-chip memory compared to access energy from the on-chip memory and computation energy, NN accelerators aims to minimize the off-chip memory accesses by exploiting the memory hierarchy and data-reuse from them.

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,

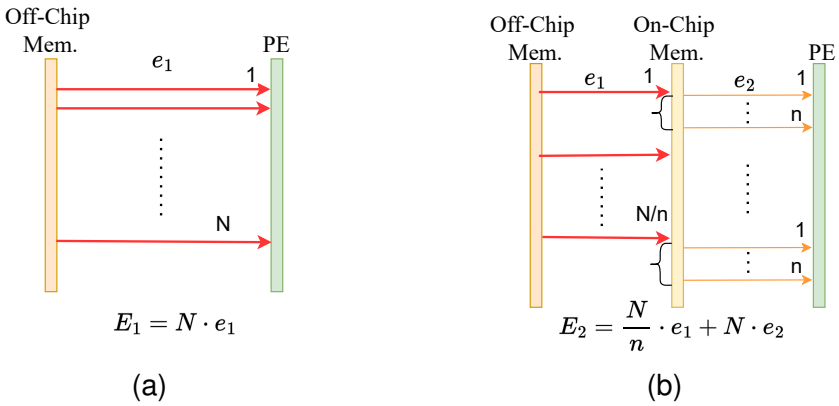


Figure 2.1: Memory accesses and energy estimates for accessing data (a) always from the off-chip memory. (b) from two-level of memory hierarchy while reusing the data from on-chip memory.

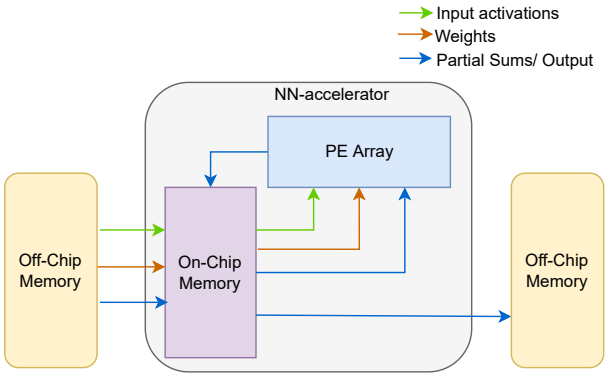
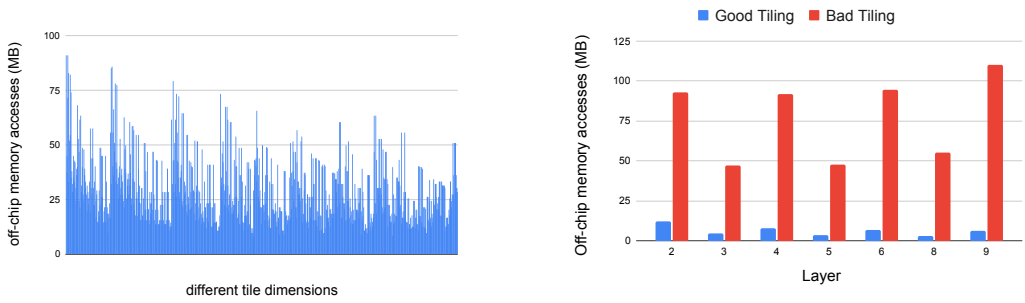


Figure 2.2: Read/Write of inputs, weights and partial/final outputs from different level of memories.

the large layer data is partitioned into small tiles. These tiles are fetched from the off-chip memory, repeatedly to compute the final output sum. Figure 2.3 shows the impact of tile dimensions on off-chip memory accesses for a popular CNN, VGG16, for a given tile scheduling scheme. Figure 2.3a shows using different tile dimensions results in different volume of off-chip memory accesses. A good tile dimension can reduce the off-chip memory accesses upto 90% compared to a bad tile dimension as shown in Figure 2.3b for different layers of VGG16. The tile dimensions and the order in which these tiles are processed (scheduling of the tile operations) significantly impact the volume of data reuse and, thus, the overall energy consumption and throughput of DNN accelerators.

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, etc.),



(a) (b)

Figure 2.3: Off-chip memory accesses for 64KB on-chip memory (a) off-chip memory accesses using different tile dimensions for a single layer. (b) Impact of good and bad tile dimensions on different Conv. layers of VGG16.

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 type, 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 given PE array. 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.

The off-chip memory access of a partitioned 3D data can be computed as follows

$$\mathbb{B}_{3D} = \sum_{t=1}^{N_{tiles}} (\mathbb{B}_t \times r) \tag{2.2}$$

where N_{tiles} is the number of tiles. r and $\mathbb{B}_t$ are the trips count and the number of bytes accessed from off-chip memory for the t^{th} tile, respectively.

The analytical framework computes the off-chip memory accesses of a given 3D data using the above methodology while considering the data resolution and architectural constraint. 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. Analytical framework implements models for different layer-types and data reuse schemes to precisely compute the memory accesses and data access energy.

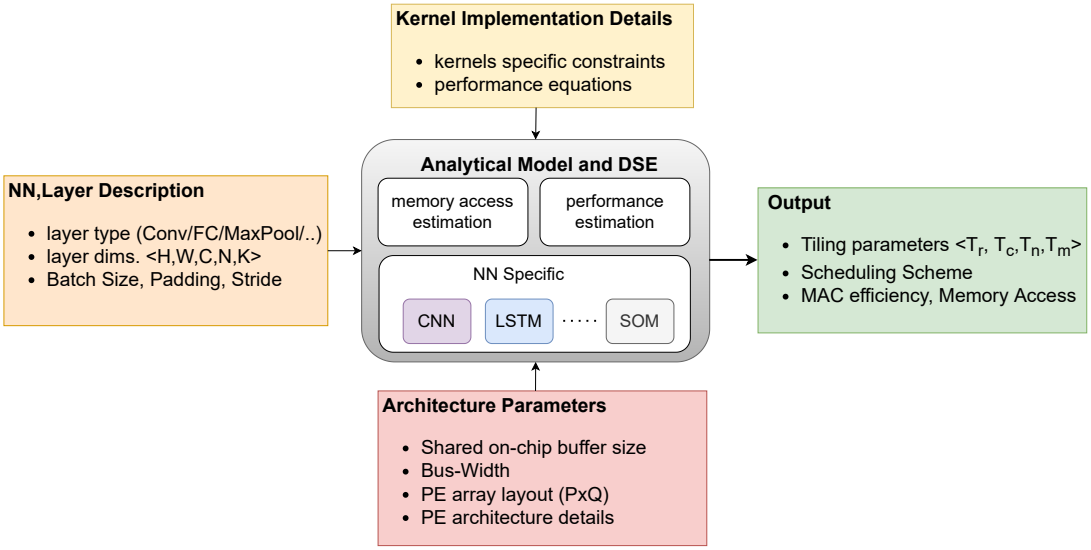
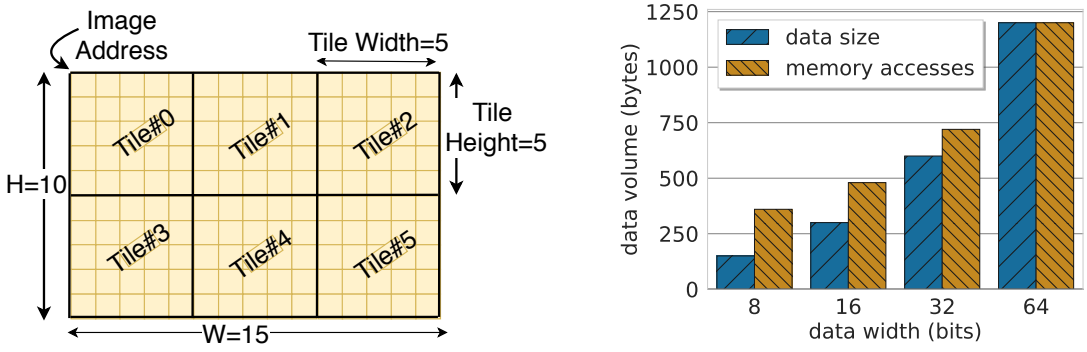


Figure 2.4: Analytical framework to estimate the performance, off-chip memory accesses and energy of DNNs.

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

Fig. 2.5a shows a 2D image 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. Table 2.1 shows the sizes and off-chip memory accesses of each tile 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.

Fig. 2.5b shows the size ($H \times W$) and off-chip memory accesses of the image shown in Fig. 2.5a, for different data bit widths. For smaller data bit width, the ratio of memory accesses to the data size is higher. Modern NN accelerators use wide memory bus to improve the

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

memory bandwith and 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.

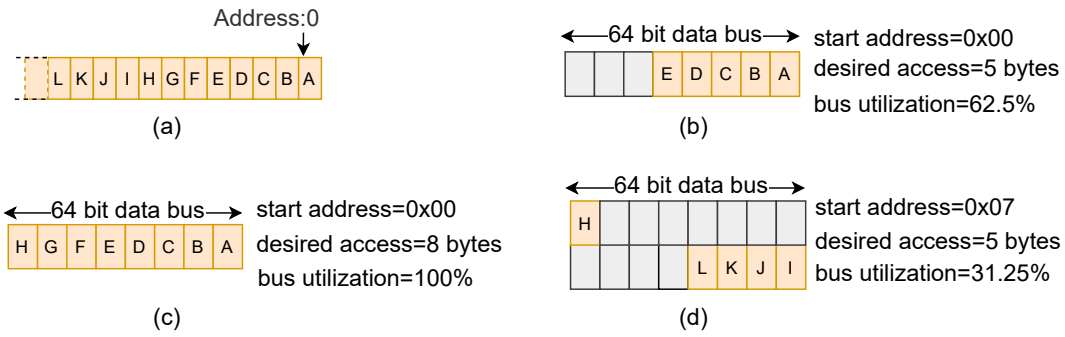


Figure 2.6: Off-chip memory accesses on 64-bit wide data bus

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. Fig. 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 Fig. 2.6c. The unused byte lanes do not carry any useful data, but they 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.

To analyze the effect of bus width and address alignment, we express 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).

2.1.1 Bus Width

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

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

2.1.2 Address Alignment

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 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

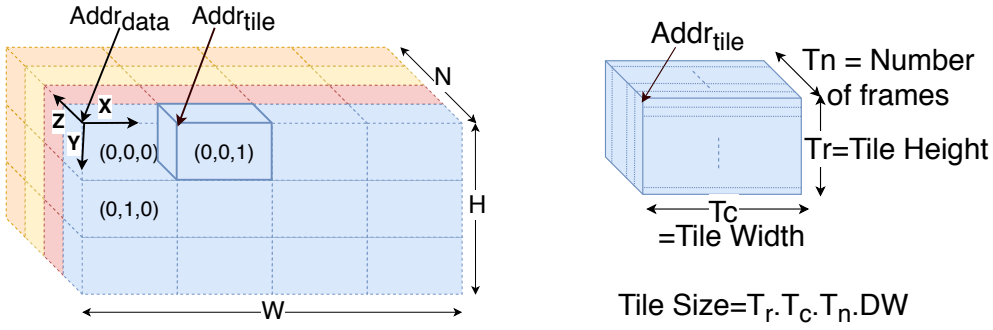
$$\mathbb{B}(Addr, l, BW) = (\lceil \frac{Addr + l}{BW} \rceil - \lfloor \frac{Addr}{BW} \rfloor) \cdot BW \tag{2.3}$$

2.2 Off-Chip Memory Access of 3D Data

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 Fig. 2.7a and Fig. 2.7b. Each tile is identified by index (x, y, z) where $x, y, z \in \mathbb{Z}$ and

$$0 \leq x < \lceil \frac{W}{T_c - \delta} \rceil, 0 \leq y < \lceil \frac{H}{T_r - \delta} \rceil, 0 \leq z < \lceil \frac{N}{T_n} \rceil \tag{2.4}$$

δ is the number of elements or rows overlapping between adjacent tiles.



(a) A 3D data partitioned into tiles

(b) A 3D tile

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

Algorithm 1 BW Aware off-chip memory access

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** $\langle T_c^1, T_r^1, T_n^1\rangle$

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 δ 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 δ in lines 5–7. Using the indices of the first element it computes the address of the tile in line 8, and 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. The address of the tile in line 8 is computed by the function GetTileAddr using equation 2.5 below

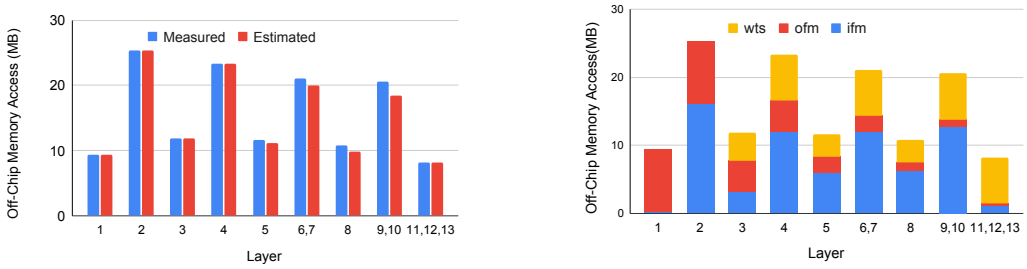
$$GetTileAddr(Addr_{data}, c, r, n, \langle W, H, N\rangle) = Addr_{data} + c + r \cdot W + n \cdot W \cdot H \tag{2.5}$$

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 is 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 2.3.

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 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.

Figure 2.8a shows the comparison between the off-chip memory accesses estimated by the analytical framework and measurements performed on Xilinx FPGA for different layers of varying shapes and sizes of a popular CNN, VGG16. The experimental results on popular CNNs, AlexNet, and VGG16, show that the difference between estimated and measured off-chip memory accesses is less than 4%. The framework is also helpful in analyzing the layer-wise distribution and breakdown of memory accesses, as shown 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 analytical framework and measured on hardware. (b) Breakdown by data type using analytical framework.

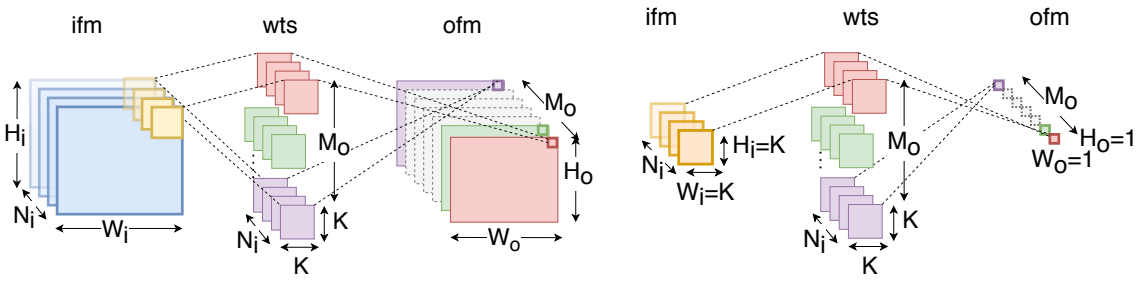
Chapter 3

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 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 last three layers are FC. Similarly, AlexNet has five CLs, followed by 3 FCLs. Each CL and FCL layer takes 3D input frames (*ifm*) and applies filter weights (*wts*) to compute output frames (*ofm*). 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. Pooling layer helps in reducing the activation sizes and thus the number of parameters in subsequent layers. CL and FC layer computations involve several parameters. The computations of a CL and an FCL are illustrated in Figure 3.1a and 3.1b, respectively.

CNN accelerators have limited on-chip memory size. Layer activations and parameters sizes are too large to fit into the on-chip memory. To peform the computations 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



(a) Convolution Layer (b) Fully Connected Layer

Figure 3.1: Convolution and fully connected layers

large arrays into smaller tiles to increase the data locality and ensures that data fits into smaller memories. Fig. 3.2 shows a layer's data stored in off-chip memory and its tiles in the accelerator's on-chip buffer.

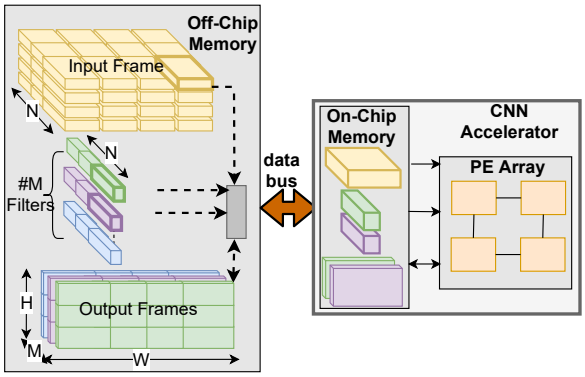


Figure 3.2: CNN layer tiles in off-chip and on-chip memory

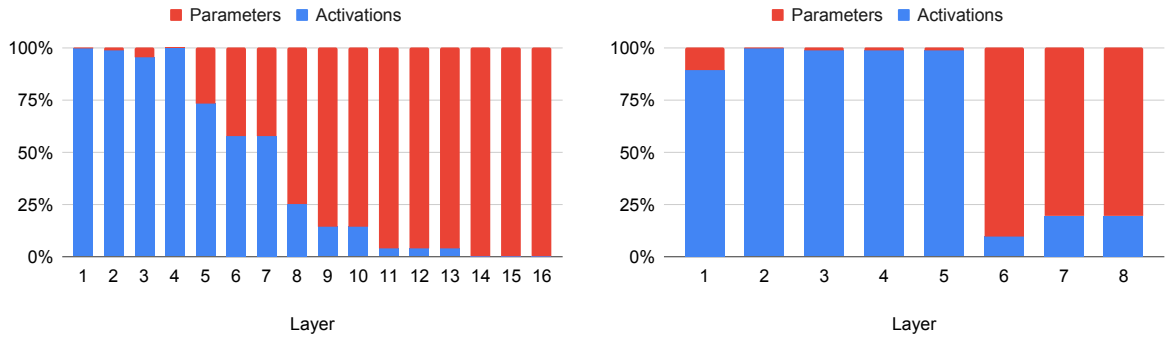
Pseudo code of tiled version of a convolution layer is shown in Listing 3.1. The outer 5 loops labelled as L_D, L_H, L_W, L_M, L_N selects the tiles of the data and inner loops perform computations on the selected tiles. Order of the outer loops decide the sequence in which different tiles of data are 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 innermost loop L_M uses the same *ifm* data. In all such loop orderings, load statement of *ifm* tile can be moved before the innermost loop L_M and reused in all its iterations [49]. The movement of ifm tile loading, reduces the number of number of off-chip accesses of ifm tile by a factor of $\frac{M}{T_m}$. This ordering scheme exploits the reuse of ifm data and referred 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 referred to as output reuse oriented (ORO). The remaining $3 \times 4!$ permutations in which loops L_D, L_R, L_C appears as the inner loops, exploits the weight data reuse and referred to as Weight Reuse Oriented

```
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

scheme(WRO). The amount of data reuse in different data reuse scheme varies with layer shape.

CNN's layers have varying shapes. Fig. 3.3 shows the parameters and activation proportions in convolution (CL) and fully connected layers (FC) of VGG16 and AlexNet. First few layers have large volume of activations (*ifm* and *ofm*) and last few CLs and FCLs have large volume of parameters(*wts* and *biases*). Scheduling schemes which optimizes the off-chip memory acccesses of activations will work well in first few layers but may not work well for deeper layers, which have large volume of *wts*. The scheduling scheme optimal for one layer may be suboptimal for other layers. Therefore, each layer need to be analyzed here.



(a) VGG16 (b) AlexNet

Figure 3.3: Parameters and activations proportions in CL and FC layers of CNNs.

3.1 Related Work

Zhang et al. [49] used loop tiling to optimize the off-chip memory accesses. They expressed the off-chip memory access as a function of tile dimensions and layer shape and determined optimal tile dimensions by enumerating all the legal tile dimensions. To reduce the hardware design complexity, they determined a global optimal tile dimension and used a common data reuse scheme for all the layers. Due to varying layer shapes, the optimal tile dimension and data-reuse scheme for different layers vary. Li et al. [28] proposed a layer-wise adaptive data partitioning and scheduling scheme to overcome this. However, their approach ignored the architectural parameters and address alignment and assumed that all tiles of the same dimensions have the same off-chip memory accesses. With this assumption, the tile dimensions determined by their approaches are suboptimal.

3.2 Off-Chip Memory Accesses of CNN Layers

3.2.1 Off-chip memory access of CL

A CNN accelerator accesses 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 expressed as the rows of the matrix $\mathbf{R}$ in Equation 3.1, where columns represent *ifm, ofm*, and *wts* trip counts.

$$\mathbf{R} = \begin{bmatrix} \mathbf{r}_{iro} \\ \mathbf{r}_{oro} \\ \mathbf{r}_{wro} \end{bmatrix} = \begin{bmatrix} D & (2\lceil\frac{N_i}{T_{n_i}}\rceil - 1)D & \lceil\frac{H_o}{T_{r_o}}\rceil\lceil\frac{W_o}{T_{c_o}}\rceil D \\ \lceil\frac{M_o}{T_{m_o}}\rceil D & D & \lceil\frac{H_o}{T_{r_o}}\rceil\lceil\frac{W_o}{T_{c_o}}\rceil D \\ \lceil\frac{M_o}{T_{m_o}}\rceil D & (2\lceil\frac{N_i}{T_{n_i}}\rceil - 1)D & 1 \end{bmatrix} \tag{3.1}$$

First we compute the off-chip memory accesses for one trip of each data type using Algorithm 1 as below

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

where $\delta = (K - S)$ is the number of overlapping elements between adjacent ifm tiles, K is the filter size, and S is the stride. $\mathbb{B}_{ifm}$, $\mathbb{B}_{ofm}$, and $\mathbb{B}_{wts}$ are 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, which are related to *ofm* layer shape and tile dimensions as below

$$\begin{gathered} H_o = (\frac{H_i + 2 \cdot P - K}{S} + 1),\ W_o = (\frac{W_i + 2 \cdot P - K}{S} + 1) \\ T_{r_o} = \frac{T_{r_i} - K}{S} + 1,\ T_{c_o} = \frac{T_{c_i} - K}{S} + 1 \end{gathered} \tag{3.2}$$

where P is the padding. The total number of bytes accessed for jth reuse scheme ($\mathbb{B}_j$) can be expressed as following sum

$$\mathbb{B}_j = \mathbf{r_j} \cdot \begin{bmatrix} \mathbb{B}_{ifm} & \mathbb{B}_{ofm} & \mathbb{B}_{wts} \end{bmatrix}^T \tag{3.3}$$

where $\mathbf{r}_j$ is the row vector of the matrix $\mathbf{R}$ for the jth scheme.

3.2.2 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. Volume of the tiles is given by equation Equation 3.4 below

$$\begin{bmatrix} V_i \\ V_o \\ V_w \end{bmatrix} = \begin{bmatrix} T_{c_i} \cdot T_{r_i} \cdot T_{n_i} \\ T_{c_o} \cdot T_{r_o} \cdot T_{m_o} \\ K^2 \cdot T_{n_i} \cdot T_{m_o} \end{bmatrix} \tag{3.4}$$

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

$$\begin{aligned} &(V_i + V_w + V_o) \cdot DW \leq buffSize \\ &0 < T_{c_o} \leq W_o,\ \ 0 < T_{r_o} \leq H_o \\ &0 < T_{n_i} \leq N_i,\ \ 0 < T_{m_o} \leq M_o \end{aligned} \tag{3.5}$$

Determining the tile dimensions which minimize the off-chip memory accesses, expressed by equation Equation 3.3, is a constraint optimization problem. The number of bytes accessed from off-chip memory using jth reuse scheme $\mathbb{B}_j$ (Equation 3.3) and the constraints (Equation 3.5)

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.

3.2.3 Off-chip memory access of FCL

The computations of FCLs are 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 3.2 are $\langle 1, 1, M_o \rangle$ and $\langle 1, 1, T_{m_o} \rangle$.

For FCLs, trips count of different data reuse schemes can be computed using equation 3.1and the number of bytes accessed from off chip memory ($\mathbb{B}^{FC}$) using Equation 3.3 by applying the layer shape constraints. The constraints on tile dimensions are given by Equation 3.5. T_{n_i} and T_{m_o} are the unknowns to be determined in Equation 3.3. Determining the optimal tile dimensions for FCLs is a constraint optimization problem, similar to CL.

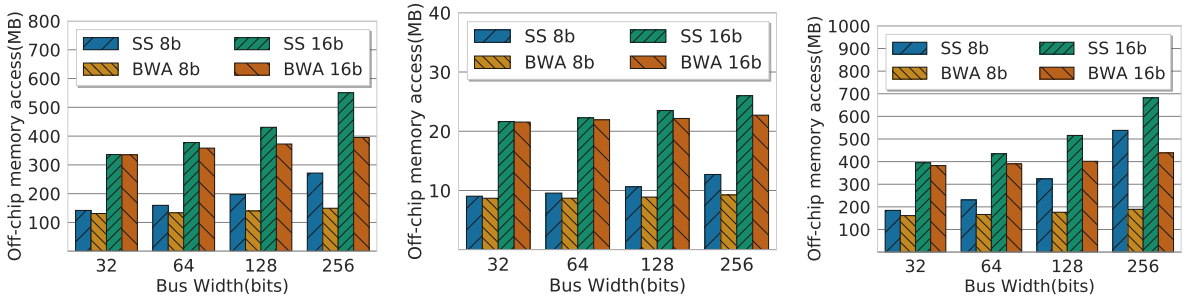
3.3 Implementation and Results

3.3.1 Implementation

The optimal tile dimensions 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 the model that computes $\mathbb{B}$ of the CLs using the BWA approach (Algorithm 1) and finds the optimal tile dimensions. 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 width. It took less than 60 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) using the Xilinx tools. Our validation code is implemented 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 provide the option to integrate the AXI Performance Monitor (APM) IP [46], which captures the real-time performance metrics like bus latency, amount of memory traffic for connected AXI interfaces. The target platform is ZedBoard, working at 100MHz frequency



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

Figure 3.4: Off-chip memory access of convolution layers for 8 and 16 bits data width. BWA: Bus Width Aware, SS: SmartShuttle

and off-chip memory (DRAM) is accessed using 64 bits AXI bus. Our validation code 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 [38], AlexNet [26], and ResNet [22] having 8, 16, and 50 layers, respectively. These CNNs have varying shapes and use filters of dimensions 1×1, 3×3, 5×5, 7×7, and 11×11. 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

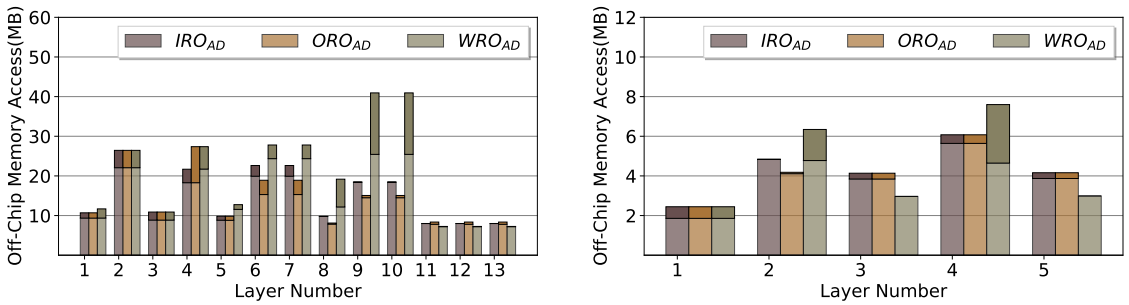
We have implemented the SmartShuttle (SS) [28] approach to compare the results with our bus width aware (BWA) approach. SS statically determines the partitioning and scheduling scheme for each layer. It performs better than strategies that use a fixed tile dimension and data reuse scheme for all layers, e.g., Eyeriss [9] and Zhang [49].

3.3.5 Results

3.3.5.1 Impact of Bus Width on memory access of CLs

Fig. 3.4 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. For a wider data bus, $\mathbb{B}$ is more. BWA approach reduces the impact of bus-width on off-chip memory accesses, as it selects the tile dimensions considering the bus width and address alignments. Whereas, tile dimensions selected by SS remains the same, irrespective of bus width of the accelerator, which results in a large value of $\mathbb{B}$. As shown in Fig. 3.4, 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



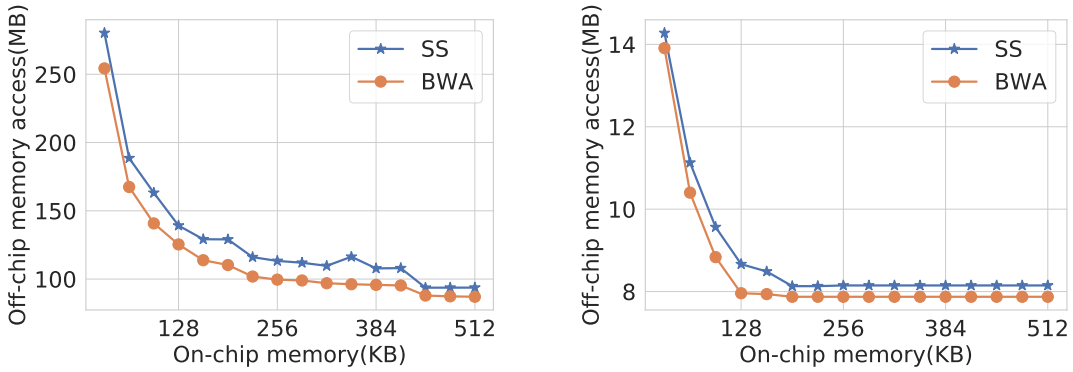
(a) VGG16 CLs (b) AlexNet CLs

Figure 3.5: Layer wise off-chip memory access for IRO, ORO and WRO schemes.

Figure 3.5 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. In the first few layers, IRO and ORO perform better than the WRO scheme, 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. The BWA approach performs better than the tile size based approach for all the three data reuse schemes.

3.3.5.3 On-chip Buffer Size



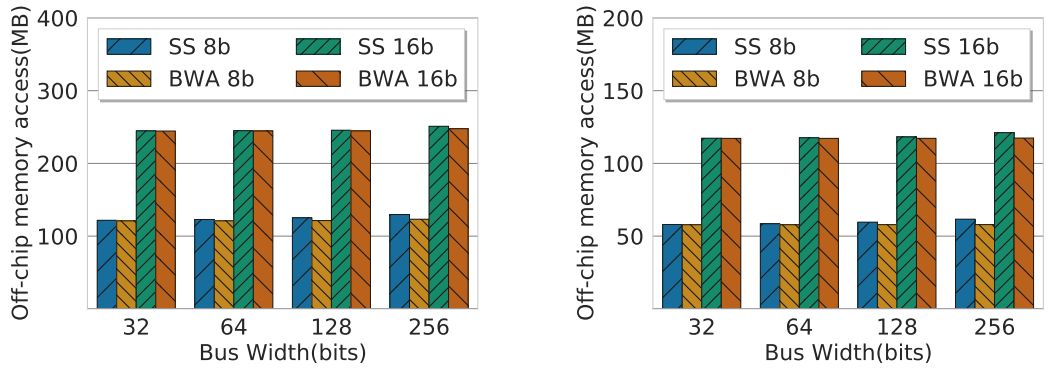
(a) $\mathbb{B}_{VGG16}$ (b) $\mathbb{B}_{AlexNet}$

Figure 3.6: Off-chip memory access for varying on-chip buffer sizes. BWA: Bus Width Aware, SS:SmartShuttle

Fig. 3.6a and Fig. 3.6b compares the number of bytes accessed from off-chip memory ($\mathbb{B}$) for different on-chip buffer sizes ($buffSize$) for VGG16 and AlexNet, respectively. Large on-chip buffer can accommodate larger tiles, which reduces the tiles trip counts (Equation 3.1) and therefore the total off-chip memory accesses of the CNN (Equation 3.3). This behavior is observed for both the CNNs in Fig. 3.6a and Fig. 3.6b. Both the 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 the address alignments and bus width to reduce unwanted data transfers to optimizes the off-chip memory accesses, while SS approach ignores the architectural parameters.

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

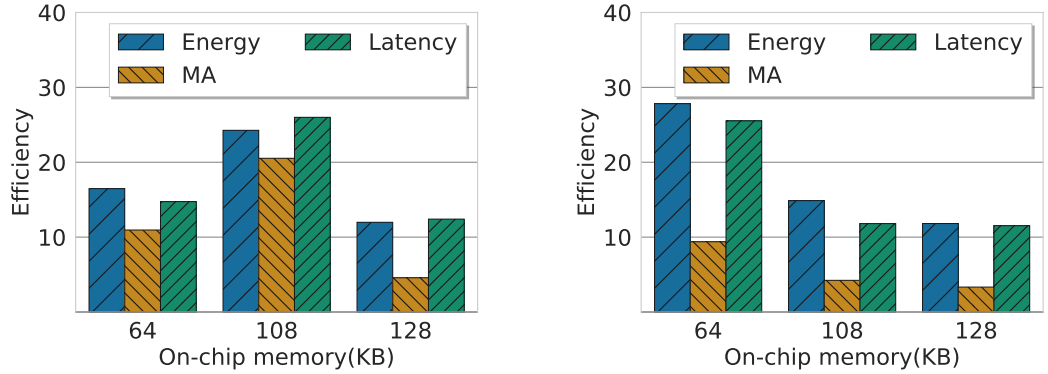
In FCLs, the height and width of tiles and data are the same. Tiles in FCL can be accessed from off-chip memory using a single transaction which results in fewer transactions in FCLs compared to CLs. Whereas in CLs, multiple transactions are required to access different rows of the tiles. Thus the impact of bus width is less on $\mathbb{B}$ of FCLs as compared to CLs. Figure 3.7a and Figure 3.7b 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.



(a) VGG16 (b) AlexNet

Figure 3.7: Off-chip memory access of Fully connected layers. BWA: Bus Width Aware, SS:SmartShuttle

3.3.5.5 Latency And Energy Analysis



(a) VGG16 (b) AlexNet

Figure 3.8: 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 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 [**?**]

$$E = P \cdot Time + \mathbb{B} \cdot E_{DDR} \tag{3.6}$$

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 [**?**].

Figure 3.8a and Figure 3.8b 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 have a significant impact on 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, and these matrix-vector multiplications are performed for a large number of 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 size 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, and b^j is the bias vector, respectively, 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 hidden state vector h_t is N, the dimensions of W^j, R^j and b^j is $N \times L$, $N \times N$ and N, respectively. N is referred to as 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 (σ).

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 and limits the data reuse.

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.

4.3 Related Work

To address the computational and energy efficiency of LSTMs and in general RNNs, several ASIC [4, 11, 42] 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, 37] 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. The proposed 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 the quantization and pruning techniques to compress the models' size. Approaches [14, 41] used 18-bit, Chang et al. [5] used 16-bit, Han et al. [20] used 12-bits 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. The proposed approach is orthogonal to the quantization techniques and can be integrated with different quantization techniques to reduce the memory accesses further.

Han et al. [20] used pruning to compress the model. However, pruning results in irregular network structure, and the sparse matrix require additional computational and storage resources and causes unbalanced load distribution. To overcome this Wang et al. [41] used block-circulant matrices representations to compress the LSTM/RNN model and to eliminate irregularities resulted from compression. Some approaches [20, 32, 33] 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 works reduced the memory accesses without effecting the accuracy of the NNs output by applying the data-reuse techniques. The matrix-vector multiplication $W^j \cdot x$ in Equation Equation 4.1, where $j \in \{i, f, g, o\}$, is independent of previous state computation. Que et al. [36] 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.([34]) 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 does not fully exploit 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 basic idea of Split And Combine Computations (**SACC**) approach and then its extension to block-wise reuse of data.

4.4.1 Basic Approach

The computation of the h_t can be expressed as shown below

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

where F is a non-linear function. q_t is computed as $W \cdot x_t + b$ and its computations are independent of previous step cell states. $S_t[k]$ is the sum of N product terms as shown below,

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

$S_t[k]$ can be computed as a sum of the following two partial sums $S_t^L[k]$ and $S_t^U[k]$

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

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

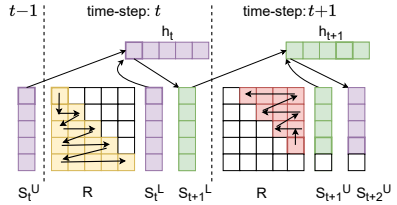


Figure 4.1: Splitting the hidden state vector computations into partial sums

Equation Equation 4.4 uses the lower-diagonal and diagonal elements of R (R^L), and Equation 4.5 uses the upper diagonal elements of R (R^U). As shown in Figure 4.1, R^L and R^U are accessed in consecutive time steps and reused in the partial sum computations of two steps. At time step t, S_t^U and h_{t-1} are the inputs from the previous time step, and R^L is reused to compute the partial sums S_t^L and S_{t+1}^L. Input S_t^U is added to S_t^L to compute h_t, and S_{t+1}^L is passed to $(t+1)^{th}$ step computations. In the same way, at time step $t+1$, R^U is reused to compute S_{t+1}^U and S_{t+2}^U. Elements of R^L are accessed from top to bottom, left to right, while elements of R^U are accessed in the reverse order to satisfy the dependencies. As shown in Figure 4.1, the proposed approach accesses the weight matrix R once, to compute h_t and h_{t+1}. Figure4.2 illustrates the compute-steps (C1 to C9) and weights accessed for computing

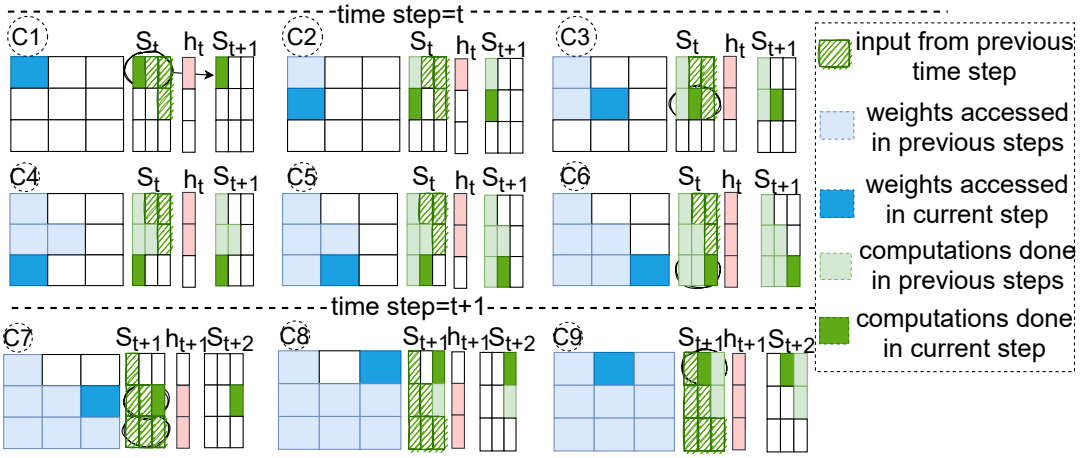


Figure 4.2: Computation of consecutive hidden state vectors h_1, h_2 and h_3 while accessing R matrix from off-chip memory.

the outputs of two time steps for N=3. The shaded rectangular blocks shows the product terms input from the previous time step (t=1). When all the product terms ($R[k][j]\cdot h_{t-1}[j]$) for the partial sum vector element ($S_t[k]$) are computed, then $h_t[k]$ is computed (shown as pink blocks in Figure4.2). The weights of R are reused for the product terms $R[k][j]\cdot h_t[j]$ for computing the partial sum S_{t+1}, using the values of h_t computed in previous or present compute-step. As shown in Figure4.2 matrix R is accessed in 9 compute-steps ($N\times N$) to

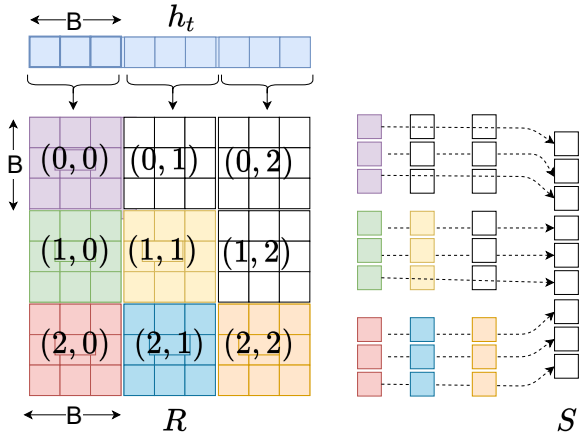


Figure 4.3: Partitions of R in $B \times B$ blocks and partial sum computations.

compute h_t and h_{t+1} and the partial sum S_{t+2} for the next time step $(t+1)$.

4.4.2 Block-wise reuse

The proposed approach partitions R into square blocks of size $B \times B$, that fits in the accelerator's on-chip memory, as shown in Figure4.3. Each block can be indexed as (r, m), where $0 \leq r, m \leq (\lceil \frac{N}{B} \rceil - 1)$. The proposed approach computes h_t in $\lceil \frac{N}{B} \rceil$ steps and at each step computes a slice of length B. The k^{th} element of r^{th} slice of h_t can be computed as following

$$h_t[B*r+k] = F\Big(\sum_{m=0}^{\lceil \frac{N}{B} \rceil - 1} S_{(r,m)}[k] + q_t[B*r+k]\Big) \tag{4.6}$$

$$S_{(r,m)}[k] = \sum_{j=0}^{B-1} R[B*r+k][B*m+j] \cdot h_{t-1}[B*m+j] \tag{4.7}$$

,where $0 \leq k \leq B-1$. The summation $\sum_{m=0}^{\lceil \frac{N}{B} \rceil - 1} S_{(r,m)}[k]$ in Equation 4.6 can be expressed as a sum of the following partial sums

$$\sum_{m=0}^{\lceil \frac{N}{B} \rceil - 1} S_{(r,m)}[k] = \sum_{m=0}^{r} S_t^L[k] + \sum_{m=r+1}^{\lceil \frac{N}{B} \rceil - 1} S_t^U[k] \tag{4.8}$$

$S_t^L[k]$ uses the lower-diagonal and diagonal blocks of R (R^L), and $S_t^U[k]$ uses the upper diagonal blocks of $R(R^U)$. The SACC approach reuses blocks of R to compute the partial sums of two consecutive time steps, similar to the approach described in 4.4.1.

Algorithm 2 describes the computations of the 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.

Algorithm 2 SACC algorithm

procedure COMPLSTMCELL(W,R,b,x_{t+1})
  $q_{t+1} \leftarrow$ 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)
      $s^B_{t+1} \leftarrow s^B_{t+1}+MXV(\mathbf{R}^B, h^B_t, 4B, B)$
      if $m = r$ **then**
        $(h^B_{t+1}, c^B_{t+1}) \leftarrow$LSTMEQNS$(v^B_x, s^B_{t+1}, c^B_t)$
        $h_{t+1}[i_s : i_e] \leftarrow h^B_{t+1}, c_{t+1}[i_s : i_e] \leftarrow c^B_{t+1}$
      $h^B_{t+1} \leftarrow h_{t+1}[m \times B : (m+1) \times B - 1]$
      $s^B_{t+2} \leftarrow s^B_{t+2}+MXV(\mathbf{R}^B, h^B_{t+1}, 4B, B)$
    $s_{t+2} \leftarrow$ 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)
      $s^B_{t+1} \leftarrow s^B_{t+1}+MXV(\mathbf{R}^B, h^B_t, 4B, B)$
      $h^B_{t+1} \leftarrow h_{t+1}[i_s : i_e]$
      $s^B_{t+2} \leftarrow s^B_{t+2}+MXV(\mathbf{R}^B, h^B_{t+1}, 4B, B)$;
    $(h^B_{t+1}, c^B_{t+1}) \leftarrow$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 $(s_{t+2}, h_{t+1}, c_{t+1})$

Both the procedures reuse the blocks of R to reduce the off-chip memory accesses.

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})

Algorithm 3 implements the LSTM equations using the partial sum vector.

4.5 Experimental Setup And Results

We have implemented LSTM layers using conventional and proposed approaches and synthesized the design using the SDSoC framework, SDx v2018.3. The design can be configured for different input vector lengths, on-chip buffer sizes, and the number of hidden units. 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 Performance Monitor (APM) IP to log the number of bytes transferred and memory access latencies for DRAM accesses.

4.5.1 Baseline

We have compared our approach with conventional and TSI-WR [34] approaches. We have used the exact implementation for off-chip memory transfer, matrix-vector multiplication, sigmoid, and tanh for both approaches to perform a fair comparison. The on-chip buffer size ($4\times B\times B$) used to store the weight matrices is also kept the same for both the approaches. Table 4.1 shows the FPGA resources utilization reported by Vivado for conventional and proposed approaches. The proposed approach requires additional on-chip memory to store the

Table 4.1: FPGA Resource Utilization(%) for B=64

	LUT	FF	Block RAM	DSP
Conv.	66.8	40.61	61.07	82.73
SAC	63.3	37.87	65.36	75.45

four partial sum vectors ($4\times N$) and four temporary vectors ($4\times B$). We have also implemented the models to compute the off-chip memory accesses for all three approaches and integrated them with 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) [39], which is widely used in natural language processing. The LSTM models are adopted from [4, 20, 32]. Each model has two LSTM layers and the parameters are described in Table 4.2.

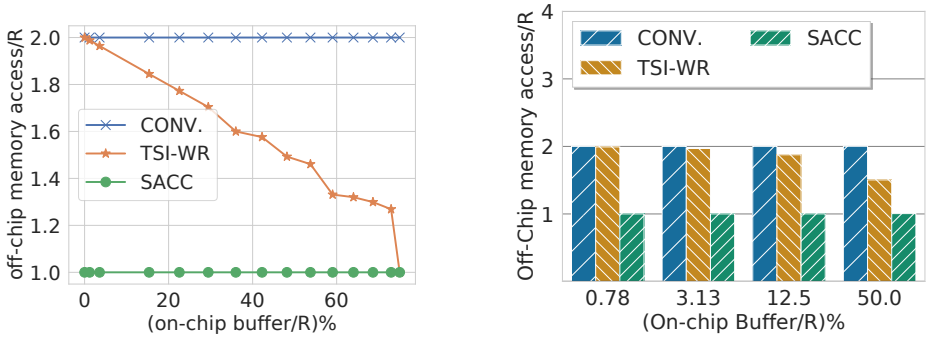
Table 4.2: LSTM Models used for experiments

Model	Input Size	#Hidden units	
		Layer 1	Layer 2
Character Level LM [4]	65	128	128
TIMIT-512 [32]	40	512	512
TIMIT-1024 [20]	160	1024	1024

4.5.3 Results

4.5.3.1 Off-Chip Memory Access

Figure 4.4 shows the off-chip memory accesses for two time-steps for conventional, TSI-WR, and SACC approaches. We have experimented for different on-chip memory sizes, when it is lesser than R. Figure 4.4a and Figure 4.4b shows the result computed using Analytical Framework and measured using our hardware implementation on FPGA, respectively.



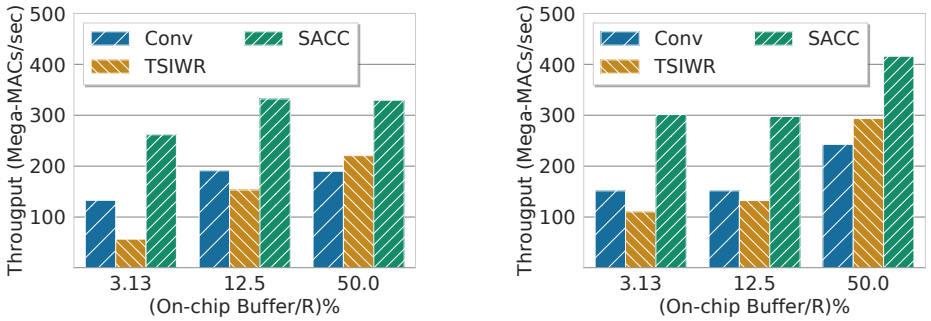
(a) (b)

Figure 4.4: 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 frameworks (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 on-chip memory. In conventional approaches, there is no reuse of these tiles for subsequent time step computations, which results in accessing full matrix R every step. For conventional approaches, the off-chip memory accesses remains same, even if on chip mem size is increased as shown by the horizontal line in Figure 4.4a. The TSI-WR approach schedules the tiles in a way 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, as shown in Figure 4.4. When the on-chip buffer to R matrix size is close to 48%, TSI-WR reduces the memory access by ≈25%. The SACC approach splits the computations and perform 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



(a) 64KB on-chip buffer size

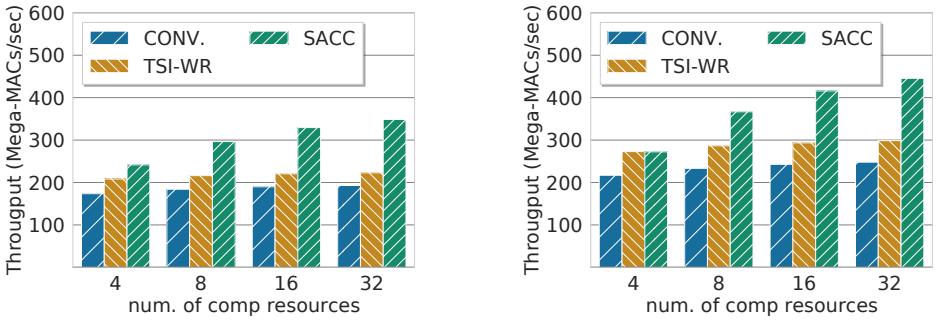
(b) 128 KB on-chip buffer size

Figure 4.5: Throughput variation with different on-chip buffer/R ratio

Figure 4.5a and Figure 4.5b compares 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 number of hidden units (N) to vary R matrix size. Increasing on-chip buffer to R matrix size ratio results in larger tile sizes and fewer number of iterations, resulting in throughput improvement, for all the 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 and insignificant 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. The proposed approach performs better than the remaining approaches for all the cases due to its significant data-reuse of weight matrix R and minimal control logic overhead. For 50% on-chip buffer to matrix size ratio, the proposed 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

Matrix-Vector multiplication dominates the LSTM accelerators' processing. The performance of LSTM accelerators is limited by the memory bandwidth. For the conventional approach, increasing the number of computing resources does not improve the performance, as shown in

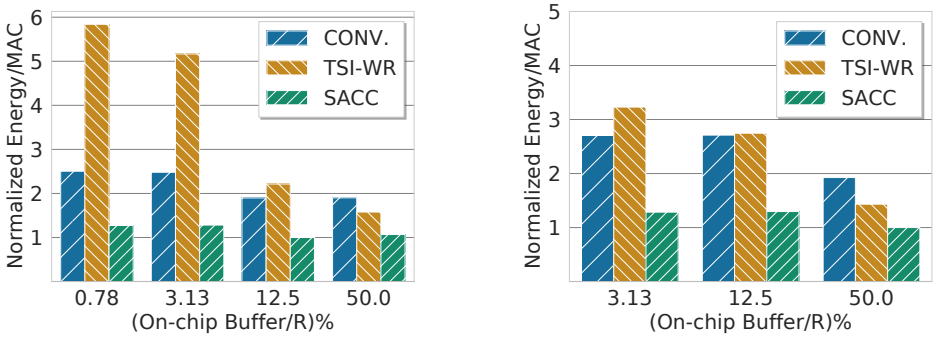


(a) 64 KB on-chip buffer size (b) 128 KB on-chip buffer size

Figure 4.6: Throughput variation of MxV with compute resources for 50% on-chip buffer/R ratio

Figure 4.6. TSI-WR approach improves the performance with increasing compute resources as it reuses the on-chip data and improves the operational intensity (ops/byte). However, the improvement in the TSI-WR approach is noticable only for the large on-chip buffer/R size ratios. In this experiment, we have compared the results when on-chip buffer to matrix size ratio of 50%. The proposed SACC approach, has better operational intensity (ops/byte) and alleviates the memory bandwith issue compared to other two approaches, which results in throughput improvement with increasing the number of parallel resources. The improvement in throughput with increased number of compute resource by the SACC approach is observed for different on-chip buffer sizes as shown in Figure 4.6a and Figure 4.6b.

4.5.3.4 Energy Efficiency Improvement



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

Figure 4.7: Energy improvement with on-chip buffer size/R

Figure 4.7a and Figure 4.7b shows 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 for higher on-chip buffer to R matrix size ratio. Off-chip memory accesses dominates the overall energy consumption. The proposed SACC approach outperforms the other two approaches for all the cases as it reduces the off-chip memory accesses. 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 enormous growth in number of Edge AI applications, there is a pressing need of efficient execution of these alogrithms on edge devices. These edge devices use customized accelerators to meet the 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. The proposed 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 the state of art TSI-WR approach.

Chapter 5

Performance Improvement of SOM by using Low Bit-Width Resolution

One of the prime reasons for NNs popularlity in last decade is due to their ability to detect pattern accurately. Compared to conventional solution they scale exceptionally well because of their inherit parallelism and are faster. NNs are also very power efficient, since their power consumption during the inference (recall) mode is very low. Apart from the NN benefits that we describe before, the main strength of the NNs comes from their ability to learn the classification criteria directly form 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, the ANN can invent an algorithm automatically that otherwise would have to be developed by the bioinformatics engineers. The algorithm is defined by the parameters of the networks, as they are developed during its training. That also provides with the possibility of retaining the network in order to update the identification features such as mutations and other genomic alterations that were not known before, without changing the algorithm itself.

Prior work in [47] has introduced Self-organizing maps (SOM) for rapid genome identification. SOM uses a type of unsupervised learning called competitive ANN learning model. The model reduces the data dimensions and it clusters similar data together [25]. A trained SOM network does not require to go 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 have been proposed for synchoros VLSI design, custom FPGA and GPUs [30, 35, 47]. Another important aspect of SOM and other NNs is their robustness. NNs have been proven to work 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 it has 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 is able to achieve better energy efficiency by trading off the quality (e.g., accuracy) and effort (e.g., energy) of computation is approximate computing [51]. Many modern applications, such as machine learning and signal processing, are able to produce results with acceptable quality despite most of the calculations being computed imprecisely [48]. 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 [31, 50]. Zhang et.al. [50] report less than 5% of quality loss obtained by simulation of the real hardware implemented in a 45nm CMOS technology.

Representing data with reduced bit-widths by trading off accuracy is one of the popular low-power strategy. The benefit of using reduced bit-width is improved energy performance. This is 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, as well as 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 of critical importance. 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 in [47] which has been used to recognize bacterial genomes.

As shown in figure 5.1, a SOM network is organized in a circle to better match genomic data. Each SOM network hosts N neurons. Each neuron has a weight vector which has the same size as the training vector, assume it's M. Therefore, the weight matrix for the whole SOM network would be $W = M \times N$. Input sequences, either for training or for inference, are vectors of size M. Each element in those input vectors represents a nucleotide.

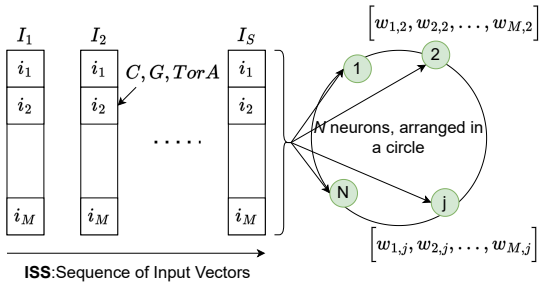


Figure 5.1: SOM based Genomic Identification

The actual training and inference processes are described in algorithm 4. The objective is to embed genomic features of a specific bacteria into a SOM. 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.

The training phase is for a specific bacterial genome. The complete bacterial DNA sequence is chopped randomly into a set of fixed size training sequence, the IS in algorithm 4 line 0. The weight matrix is initialized by random weights. On the line 1, a parameter β, initialized to 1.0, is used to control the converging process of SOM. In line 3 and 4, the actual training process first tries to find the neuron that best matches the particular input vector. Then, weights of the neighbourhood of the winning neuron are updated based on the distance between the target neuron and the winning neuron, as shown in line 6 and 7. Finally in line 8, the parameter β is decreased by a factor. After repeating the training process for all training vectors, the SOM will learn the features of that particular bacterial genomes. 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 test phase, some test DNA fragments from one unknown bacteria are send 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 line 10. 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
1: $\beta_{min} = 0.01$; $decay\_factor = 0.99$; $\beta = 1.0$
2: **for** $I_k \in IS$ **do**
3: $dist_{min} = \min_{j=1...N}(\sum_{i=1}^{M} |I_{k,i} - W_{i,j}|)$
4: $j_{min} = j$ where $dist_j = dist_{min}$
5: **for** $j \in 1...N$ **do**
6: $dist = \frac{N}{2} - ||j - j_{min}| - \frac{N}{2}|$ ▷ toroid distance
7: $W_j = W_j - \frac{\beta}{2^{dist}}(W_j - I_k)$
8: $\beta = \min(\beta * decay\_factor, \beta_{min})$ ▷ decay β
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
9: $Inferred\_r$ is r with
10: $score$= $\min_{r=1...R}\left[\sum_{k=1}^{S} \min_{j=1...N}(\sum_{i=1}^{M} |I_{k,j} - W_{r,i,j}|)\right]$

5.3 Low bit-width FPGA Design of SOM

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

A custom semi systolic array was hand crafted, for 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 input vector. This neuron is referred as winning neuron of the network. The design outputs the distance between the input vector and 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. Figure 5.2 shows a high level schematic of the FPGA implementation

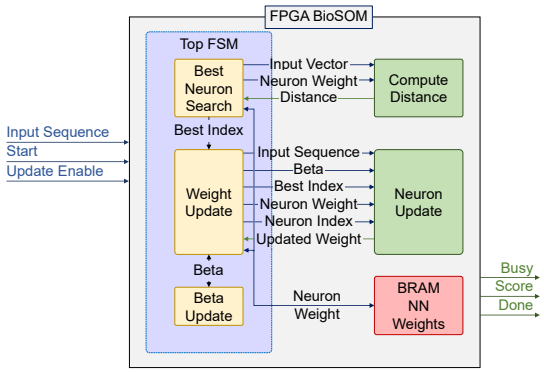


Figure 5.2: Hardware Module for BioSOM.

of BioSOM and illustrates the key components in the design. The input is a n-bit vector. Each pair of bits in the input represents one of nucleotide A,C,G or T. Thus a 16-bit word contains 8 symbols.

The Neural Network weights are stored in 2-dimensional array in BRAMs. 1^{st}-dimension represent neurons in the network and 2^{nd}-dimension holds the weights of each neuron. Each neuron has 8 weights and each weight is stored as a fixed-point number. Bit width analysis is performed by varying the number of bits (8, 12, 16, 24 and 32) used to represent the weights.

During testing phase, only *Compute Distance*, Figure 5.3, is enabled (*Update Enable=0*). This component returns the distance of a neuron from input weights. Using this distance, winning neuron of the network is identified. This component has pipe-lined implementation with an Initiation Interval (II) equal to 1. Each stage of the pipeline computes the difference between one input weight and corresponding neuron weight and updates the input partial sum. Output of each stage is fed as partial sum into next stage. Last stage output is the distance between the input and neuron.

During training phase *Neuron Update* component is enabled by setting (*Update Enable=1*). The weights of the Neurons are updated using the distance output of the *Compute Distance* module. The *Neuron Update* component is also pipe-lined design with II=1. The stages of pipeline are shown in Figure 5.4.

Clearly, the *Neuron Update* can only start after *Compute Distance* has completed finding closest neuron. So these two components can execute sequentially for a given input vector. However, while *Neuron Update* is updating the weights for an *Input Vector=i*, *Compute*

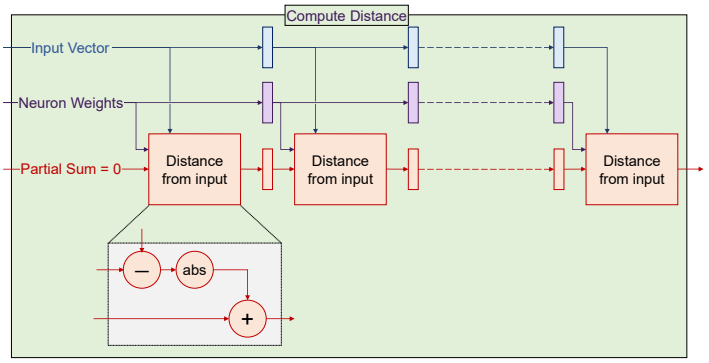


Figure 5.3: Compute Distance Module.

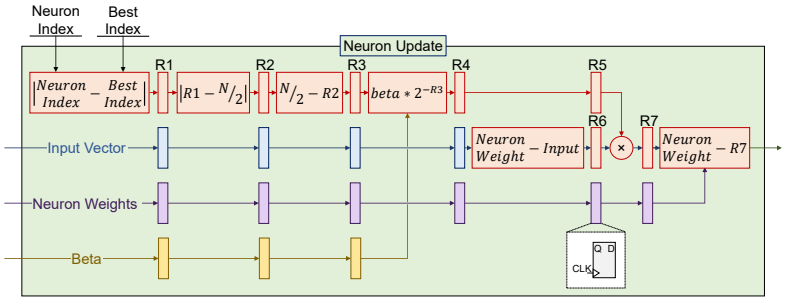


Figure 5.4: Neuron Update Module.

Distance can start its operations for next *Input Vector=i+1*. *Compute Distance* and *Neuron Update* components can overlap their execution, processing different input vectors at same time. This design has overall $II = 23 + N$ cycles, where N is number of neurons in the network.

5.4 Experimental Results And Analysis

In this section we present the results from 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 presents 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 expect that the SOM network for bacterial identification to suffer from accuracy degradation. A MATLAB simulation model was created to analyze the accuracy loss when using fixed-point implementation. The experiment used a scaled down version of 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 10 different bacteria DNA sequences. Each SOM network has 100 neurons inside, and each neuron has 20 weights. We trained the networks by 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}):

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

The *classification error* 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 error and quantization error are not completely independent, but they reveal the different aspects of fixed-point implementation accuracy. Quantization error shows the difference between fixed-point representation and golden double point reference. It doesn't 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 error and the classification error 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%, 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.

We like to note that each SOM was trained with the specific fixed-point resolution and that the quantization error shows the difference of the *score* variable, see algorithm 4, between the

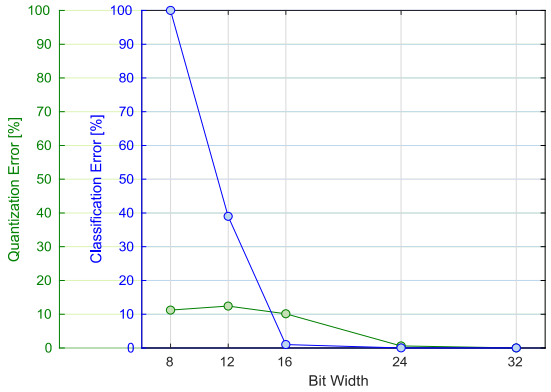
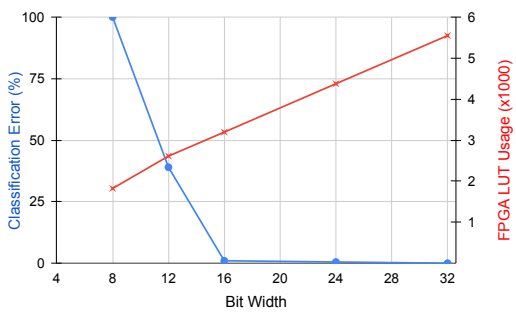


Figure 5.5: Quantization error compare to the identification error

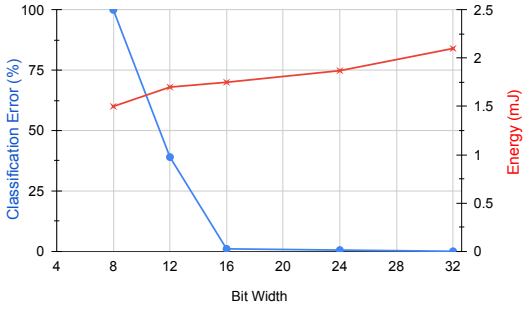
fixed-point and the double precision. The quantization error is very important for the *score* variable of the SOM, since this variable is the one that decides which is the winning SOM, i.e. which SOM matches the input DNA sequence. It is important to guarantee sufficient bits for the *score* of each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

5.4.2 FPGA experimental setup

In this section we present the experimental results of the FPGA implementation presented in section 5.3. The FPGA design is implemented with Vivado v.2016.4 used for synthesis and analysis of 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.



(a) FPGA LUT utilization



(b) energy of FPGA implementation

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

The area and power numbers for different weight resolutions are extracted from the reports generated by 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.

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 results are summarized in the Figure 5.6a and 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% of the SOM accuracy.

5.5 Summary

In this work we explore the design space of a self-organizing map (SOM) used for rapid and accurate identification of bacterial genomes. This is an important health care problem because even in Europe, 70% of prescriptions for antibiotics is wrong. The SOM is trained on Next Generation Sequencing (NGS) data and is able to identify the exact strain of bacteria. This is in contrast to conventional methods that require genome assembly to identify the bacterial strain. SOM has been implemented on FPGA and shown to have better computational efficiency compared to GPUs. To further lower the energy consumption, we exploit the robustness of SOM by successively lowering the resolution to gain further improvements in efficiency and lower the

implementation cost without substantially sacrificing the accuracy. We do an in depth analysis of the reduction in resolution vs. loss in accuracy as the basis for designing a system with the lowest cost and acceptable accuracy using NGS data from samples containing multiple bacteria from the labs of one of the co-authors. The objective of this method is to design a bacterial recognition system for battery operated clinical use where the area, power and performance are of critical importance. We demonstrate that with 39% loss in accuracy in 12 bits and 1% in 16 bit representation can yield significant savings in energy and area.

In this work, we implemented FPGA design for SOMs. The mapping support different bit width to enable trade-offs between accuracy and computation cost. Experiments has shown the big design space, resulting in different cost in terms of timing, area as well as energy, all of which are affected significantly by the fixed-point format representation. Through the experiment, we conclude that the SOM network with 16-bit fixed-point representation implemented on FPGA, has better benefits compared to other fixed-point formats with more bits. And 16-bit has acceptable classification error. Format with bits more than 16 no longer add any benefit to lower the classification error.

In future works, we plan to apply more sophisticated methods to scale down the bit width without losing too much accuracy. For example, the weights can be dynamically scaled after several epochs of training when the current fixed point format is not suitable anymore. We are confident that, with such approximate computing techniques, we could possible reduce the resolution to 8 bits with acceptable loss of accuracy and, by extension, the implementation cost of SOM networks on hardware.

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] 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.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

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

[36] 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.

[37] 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.

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

[39] 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.

[40] 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.

[41] 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.

[42] 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.

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

[44] S. Williams, A. Waterman, and D. Patterson. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009.

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

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

[47] 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.

[48] 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.

[49] 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.

[50] 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.

[51] 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.