UNIVERSITY OF CALIFORNIA SAN DIEGO

Design and Optimization of Hardware Accelerator Design

A thesis submitted in partial satisfaction of the
requirements for the degree of
Master of Science

in

Electrical Engineering (Electronic Circuits and Systems)

by

Navateja Alla

Committee in charge:

Professor Hadi Esmaeilzadeh, Chair
Professor Truong Nguyen, Co-Chair
Professor Farinaz Koushanfar

2020

The Thesis of Navateja Alla is approved, and it is acceptable in quality and form for publication on microfilm and electronically:

Co-Chair

Chair

University of California San Diego

2020

EPIGRAPH

If you don't take risks
you can't create a future.

Monkey D. Luffy

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

ACKNOWLEDGEMENTS

I would like to acknowledge Professor Hadi Esmaeilzadeh for his support as the chair of my committee. Through multiple drafts and many long nights, his guidance has proved to be invaluable. I would want to thank him for introducing me to the concept of Hardware Acceleration Design. His class on Accelerator Design for Deep Learning (CSE 240D) and Advanced Microarchitectures (CSE 240C) has kindled interest in me for pursuing research in this area. Conversion of intricate details of the machine learning algorithms into hardware has kept my curiosity going. Working in his lab Alternative Computing Technologies (ACT) Laboratory has immensely helped me both to nuture my self with the concepts of advanced hardware acceleration and also with basics of hardware design. I would also like to thanks Soroush Ghodrati who has been my mentor throughput the process and guided me in the right direction and helped me with micro details of the work.

In addition I would like to thanks Professor Ryan Kastner for helping me with concepts of FPGAs and hardware acceleration on them, Professor Cheolhong An for introducing me to GPU parallel programming and Professor John Eldon for helping me with concepts of VLSI and RTL design, Manasa Alla for helping me with the documentation. All the chapters in this journal have been taken from the papers suggested as part of hardware acclerator courses at UCSD and Georgia Tech. All the process that involve the development of each micro architectural details have been presented from them which have been cited at the end of the thesis.

Chapter 4, in part, has been submitted for publication of the material as it may appear in MICRO 2020. The thesis author is one of the co-author in this paper. This paper ’PLANARIA’ is coauthored with G. Soroush, A. Byung Hoon, K. Joon Kyung, K. Sean, Y. Brahmendra, S. Hardik, M. Alian, K. Nam Sung, E. Hadi. I would like to thank them for allowing me to be a contributor for their work. Finally, I would once again thank Professor Hadi and Soroush Ghodrati for allowing me to be a part of the LAB and ongoing work.

VITA

2011-2016 B.E. in Electronics and Communication Engineering, Birla Institute of Technology Science, Pilani

2011-2016 M.Sc. in Chemistry, Birla Institute of Technology Science, Pilani

2018-2020 M.Sc. in Electrical Engineering, University of California San Diego

ABSTRACT OF THE THESIS

Design and Optimization of Hardware Accelerator Design

by

Navateja Alla

in Electrical Engineering (Electronic Circuits and Systems)

University of California San Diego, 2020

Professor Hadi Esmaeilzadeh, Chair
Professor Truong Nguyen, Co-Chair

Deep Neural Networks have reinvigorated real-world applications that rely on learning patterns of data and are permeating into different industries and markets. With failure of Moore's law, recently industries shifted towards multicores which could not scale as expected leaving most of the chip in off state also called as dark silicon. this end of Dennard scaling and diminishing benefits from transistor scaling has propelled an era of Domain Specific Architectures.

Basic hardware accelerators were designed with primitive units for computations which are arranged spatially to account for locality and hence better performance. Dataflow in this spatial architecture determines the efficiency and power savings that can be obtained. An optimized dataflow that reuses all activations, weights , and partial sums has been discussed. Several optimization techniques including bit level flexibility, sparse accelerator which exploits weight and activation sparsity, analog accelerators for low power, new ISA for efficient hardware to software mapping, in memory computation to reduce the memory bandwidth requirements, approximate circuits, efficient interconnects have been discussed. Later the acceleration for training in various substrates are explored.

Finally, A novel method to handle multi tenancy is proposed. Cloud infrastructure and accelerators that offer INFaaS have become the enabler of this rather quick and invasive shift in the industry. Although multi-tenancy has propelled datacenter scalability, it has not been a primary factor in designing DNN accelerators due to the arms race for higher speed and efficiency. This paper sets out to explore this timely requirement of multi-tenancy through a new dimension: dynamic architecture fission. To that end, we define Planaria1 that can dynamically fission (break) into multiple smaller yet full-fledged DNN engines at runtime. This microarchitectural capability enables spatially co-locating multiple DNN inference services on the same hardware, offering simultaneous multi-tenant DNN acceleration. As such, it can simultaneously co-locate DNNs to enhance utilization, throughput, QoS, and fairness. We compare the proposed design to PREMA [4], a recent effort that offers multi-tenancy by time-multiplexing the DNN accelerator across multiple tasks. We use the same frequency, the same amount of compute and memory resources for both accelerators. The results show significant benefits with (soft, medium, hard) QoS requirements, in throughput (7.4, 7.2, 12.2), SLA satisfaction rate (45%, 15%, 16%), and fairness (2.1, 2.3, 1.9).

Chapter 1

Introduction

1.1 Computation Requirements of Deep Learning Algorithms

Artificial intelligence is wide ranging branch of computer science concerned with building smart machines capable of performing tasks that typically require human intelligence. AI is an interdisciplinary science with multiple approaches, but advancements in machine learning and deep learning are creating a paradigm shift in virtually every sector of the tech industry. The field of Machine Learning helps develop computation models that learn the environment without explicit programming. The goal is to reach human intelligence which involves a lot of computation both in terms of hardware and software.

To this end, researchers have developed many models such as SVM, decision trees, and regression. One of the most promising models so far is artificial neural networks. Inspired by biological neurons, McCulloch and Pitts developed the first model of ANNs in 1943. Later, at the end of the 1950s, a perceptron had been proposed, raising optimism about imminent human level intelligence. However, in 1969, Minsky and Papert showed the weaknesses of the perceptron model, which discouraged further activity in ANNs. In the 70s and 80s, backpropagation had been introduced and developed for training a neural network from raw data. Later in the 90s, LeCun et al. proposed convolutional neural networks (CNNs) leading to promising results in handwritten character recognition.Even with the advent of CNNs, researchers were still relying on other approaches such as SVM or ensemble of different models to achieve the best results. The power of ANNs was revealed to researchers as the size of networks and training data sets grew. Particularly, Alexnet was a milestone that won the ImageNet competition in 2012 by reducing the error rate by almost a factor of two, compared to other approaches. This work successfully trained a multimillion-parameter network with millions of raw input images using back-propagations. Alexnet training took 6 days. Without a high-speed GPU for training, Alexnet training would have taken much longer. In other words, the computation power of today's machines is a primary driver for major advancements in the field of machine learning. Machine learning researchers have also developed

A number of techniques in the last decade to help deep networks learn, e.g., the use of shared weights, dropout, expanded inputs, better activation functions, and regularization. The Alexnet structure – a sequence of convolutional layers followed by fully-connected classifier layers that is used for image classification – has also been used in many subsequent works and rejuvenated the field of deep learning. In deep learning, multiple nonlinear layers automatically extract and abstract features from raw data for different purposes such as classification and prediction. The deeper layers in the networks combine more simple features from the earlier layers to extract more complex features and recognize complicated objects in input images. Such deep neural networks (DNNs) have recently achieved better results than human image classification. However, these outstanding results are not for free; DNNs require billions of operations per image for the simple task of classification.

Figure 1.1 shows the top-1 accuracy and the computational requirements for recent DNNs. In this figure, the trend suggests that more computation leads to higher accuracy. In addition, the computational requirement grows significantly with increase in the size of input, number of training samples, and the number of classification categories. Therefore, providing faster machines is essential. Deep networks have to be trained. The training currently takes many weeks across several high-powered GPUs in a datacenter. Once the network is trained, it is deployed on several devices (datacenter servers, self-driving cars, drones, mobile devices, etc.),

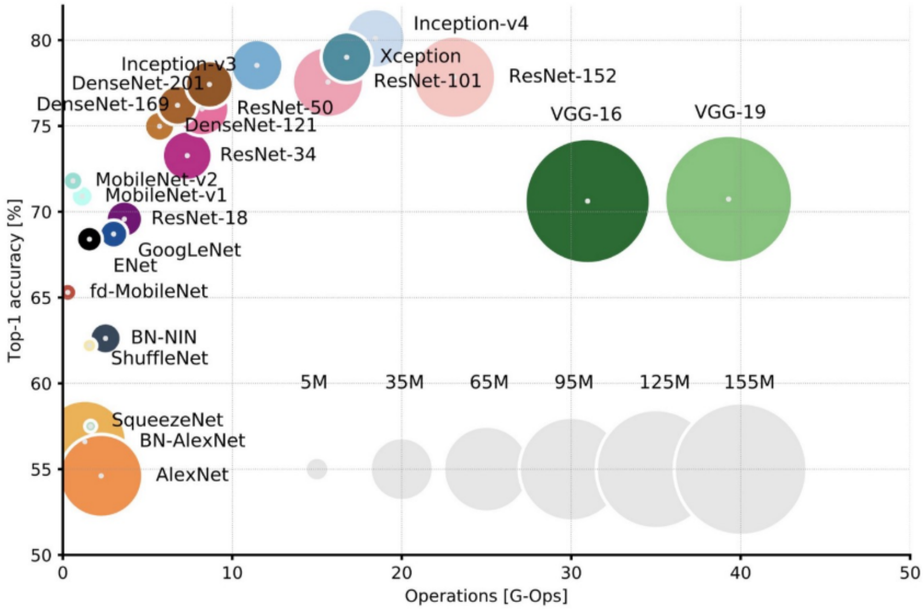


Figure 1.1. The classification accuracy vs. computation requirements (GOps) for the inference step in recent well-known image classifiers [25].

where it performs inference on billions of images every day. Faster machines are not only essential for the training operations in deep networks, they are also essential for inference operations. This dissertation focuses on both inference and training. One promising solution to the computational requirements of DNNs is hardware specialization. It is well known that custom ASICs can be up to three orders of magnitude faster than general purpose systems. Since effective neural networks have always been computation-intensive, there have been many prototypes and hardware architecture proposals. Most of this prior work focuses on digital architectures, and we review some of them in the background chapter. In this dissertation, we explore the use of analog units for DNN acceleration and make some of the first contributions in the field of in-situ analog computing for DNNs. To make the potential impact of this work clear, consider the following concrete use case. Some modern cars (and likely most cars in the future) use a variety of cameras and sensors to gather road/traffic information. Processing this information will require computers that consume several hundreds of watts and precious space. The accelerators defined in this dissertation will help process many more images (safety) with a computing system that consumes tens of watts (energy efficiency) and fits under the seat of the car.

1.2 Thesis Overview

The end of Dennard scaling and diminishing benefits from transistor scaling has propelled an era of Domain Specific architectures. Although, most recently, accelerators have made their way into edge devices(Edge TPU, NVIDIA Jetson, Apple Bionic Engine), their limited computational capacity still necessitates offloading most of the inference tasks to the cloud. Inference as a service has become the backbone of deployed applications and inference currently dominates the market and is enabled by various forms of custom accelerators, such as Google TPU, NVIDIA T4, Microsoft Brainwave, and Facebook's DeepRecSys.As demand for INFaaS scales, one solution is to increase number of accelerators in the cloud. However, this is not feasible as it is not cost effective or scalable with increasing demand for DNN service. Multi tenancy, where a single node is shared across multiple requests has been a primary enabler for the success of cloud computing in current scale. Multi tenancy not been a primary factor in the design of DNN accelerators due to recent adoption of accelerators in data centers, challenges associated with multi tenancy in accelerator like efficiently sharing the underlying hardware while enforcing strict data and performance isolation between tenants and not much research done in this area as most of the research was involved in designing the fastest accelerator both in terms on compute and memory.

The datacenter accelerator designs revealed–for instance in Google's TPU or Microsoft Brainwave tend to show results focused on running a single neural network model as fast as possible. Even the MLPerf benchmark suite keeps this single-model focus for both training and inference. But experience in cloud accelerator systems shows that keeping multiple models simultaneously resident on an accelerator has deployment benefits. Beyond just multiple customers sharing an accelerator, there is demand for multi-tenancy inside of a single application. For example, speech recognition and voice synthesis systems tend to require multiple models in deployment and can significantly benefit from multi-tenancy and co-location.

1.2.1 Thesis Statement

We explore a new dimension of multi-tenancy in the architecture design of DNN accelerators. This work presents Planaria, where the key idea is dynamically fissioning the DNN accelerator at runtime to spatially co-locate multiple DNN inferences on the same hardware. To that end, the paper makes the following contributions. First, This paper introduces and explores the dimension of dynamic fission in DNN accelerators. This innovation enables simultaneous execution of multiple DNN acceleration threads to be spatially co-located on the same hardware substrate. Second, we device a microarchitecture design for dynamic fission where we devise bi-directional systolic arrays for DNN acceleration that permits flow of data in all four directions from each elements in the array. This low-cost additional flexibility expands the fission possibilities leading to significant energy reduction and performance gains. Third, To leverage architecture-level fission, the paper defines a task scheduling algorithm that breaks up the accelerator with respect to the current server load, DNN topology, and task priorities, all while considering the latency bounds of the tasks.

1.2.2 Planaria

Deep Neural Networks (DNNs) have reinvigorated real-world applications that rely on learning patterns of data and are permeating into different industries and markets. Cloud infrastructure and accelerators that offer INFerence-as-a-Service (INFaaS) have become the enabler of this rather quick and invasive shift in the industry. To that end, mostly acceleratorbased INFaaS (Google's TPU, NVIDIA T4, Microsoft Brainwave, etc.) has become the backbone of many reallife applications. However, as the demand for such services grows, merely scaling-out the number of accelerators is not economically cost-effective. Although multi-tenancy has propelled datacenter scalability, it has not been a primary factor in designing DNN accelerators due to the arms race for higher speed and efficiency. This paper sets out to explore this timely requirement of multi-tenancy through a new dimension: dynamic architecture fission. To that end, we define Planaria1 that can dynamically fission (break) into multiple smaller yet full-fledged DNN engines at runtime. This microarchitectural capability enables spatially co-locating multiple DNN inference services on the same hardware, offering simultaneous multi-tenant DNN acceleration. To realize this dynamic reconfigurability, we first devise a breakable bi-directional systolic arrays for DNN acceleration that allows omnidirectional flow of data. Second, it uses this capability and a unique organization of on-chip memory, interconnection, and compute resources to enable fission in systolic array based DNN accelerators. Architecture fission and its associated flexibility enables an extra degree of freedom for task scheduling, that even allows breaking the accelerator with regard to the server load, DNN topology, and task priority. As such, it can simultaneously co-locate DNNs to enhance utilization, throughput, QoS, and fairness. We compare the proposed design to PREMA [4], a recent effort that offers multi-tenancy by time-multiplexing the DNN accelerator across multiple tasks. We use the same frequency, the same amount of compute and memory resources for both accelerators. The results show significant benefits with (soft, medium, hard) QoS requirements, in throughput (7.4x, 7.2x, 12.2x), SLA satisfaction rate (45%, 15%, 16%), and fairness (2.1, 2.3, 1.9).

1.3 Layout of this thesis

The rest of the thesis is organised as follows. Initially the basics of neural networks and their importance in solving day to day complex issues has been discussed. Then we project why mulicores cannot be scaled in terms of frequency and power to satisfy the increasing demand leading to dark silicon and what has propelled the need for Domain Specific architectures. With advent of domain specific architectures, principles of improving the compute and memory performance is discussed. There are multiple chapters discussed in related work section that focuses on the arms race for high performance computing.

Then we discuss methods to improve datacenter performance and finally discuss about multi tenancy in DNN accelerators which is the core idea of the paper. We then build upon the work and present some extensive design space analysis performed on this design. Finally we discuss about the flexibility in DNN accelerators and talk about our novel idea of Planaria and how it is being implemented to efficiently share the underlying hardware while enforcing strict data and performance isolation between tenants. After this we discuss the future improvements that can be done to improve the performance of multi tenant DNN accelerators.

Chapter 2

Background

2.1 Neural Networks

2.1.1 Introduction

In this chapter we explain the computational requirements of deep neural networks(DNNs). DNNs are built by connecting different layers of neurons serially or in parallel, and they typically represent a direct acyclic graph (DAG) of computations. Depending on the applications, one might leverage different types of layers. In this section, we review some of the common layers, in both inference (forward path) and training phase (backward error propagation). More specifically, we review both forward and backward paths for fully-connected, convolution, pooling, sigmoid, and ReLU layers.

2.1.2 Computation Flow

Since DNNs are DAGs, the flow of computation in the inference mode is straightforward. The input data is the first layer's input and the output of each layer will serve as the input for the next layer in the graph of computation. In the case of classification, the last layer's neurons can be interpreted as the predicted chance of one classes. In the training mode, a pair of sample data and a label will be considered as the input. Similar to the inference mode, DNN receives the sample data and generates a vector of probability as its output. Then a loss function (also known as cost function) evaluates the result by comparing it with the label. The goal of training is to reduce the loss function. One can consider the entire neural network as one complex function. The goal is to reduce the sum of the loss function output for all the training samples. Assume

In this section, we reviewed common layers used in state-of-the-art DNNs. We showed that computation intensive layers such as convolutions and FC have the same type of operations in both forward and backward paths.$S = \{(x_i, l_i)\, i \in \{0,..,N-1\}\}$ is the set of training samples with N members. Also consider a neural network with M cascaded layers. We represent Layer i with a function $f_i(x)$ and its paarameter as $W^(i)$. Therefore, the output of the entire network, for the input x_i is

$$out_i = f_{M-1}(...(f_2(f_1(x_i)))...) \tag{2.1}$$

The loss value for this input is $Loss(out_i, l_i)$. The goal is to minimize the following equation.

$$L = \sum_{0}^{N-1} Loss(out_i, l_i) \tag{2.2}$$

There are multiple ways to solve this optimization problem. In the *gradient descent* approach, in each step, L is calculated and $W^{(i)}$ are updated in a direction to get closer to the local minimum. For the layer k, the $i-th$ parameter is updated using the following rule:

$$W_i^{(k)} = W_i^{(k)} - \eta \times \frac{\partial L}{\partial W_i^{(k)}} \tag{2.3}$$

In the above equation 2.3, η is the learning rate. Large η values lead to fast convergence at the risk of missing some local minimum. On the other hand, small ηs do not jump over optimum points at the cost of slower convergence. The problem with gradient descent is that for every update step, we have to calculate all out_is. Therefore, the time for each step grows linearly with the training set size. As a result, this technique is not used for large-scale DNNs in practice. Instead, *stochastic gradient descent* will be applied. In this approach, the training set is shuffled and decomposed into many small *minibatches* and gradient descent is applied to each minibatch. Therefore, the number of outputs involved in each parameter update step is a function of the number of elements in each minibatch. In practice, minibatches are much smaller than the training set. The process of training all the minibatches is called an *epoch*. Since updating parameters is based on a few samples in the minibatch, the training is carried for multiple epochs.

In each weight update step, we also need to calculate the gradient of each weight with respect to the loss function. Applying the chain rule, we can find the gradient for the functional representation of the neural networks 2.1.

$$
\begin{aligned}
\frac{L}{\partial W_i^{(k)}} &= \frac{\partial L}{\partial y_{N-1}} \times \frac{\partial y_{N-1}}{\partial W_i^{(k)}} \\
\frac{\partial y_{M-1}}{\partial W_i^{(k)}} &= \frac{\partial y_{M-1}}{\partial y_{M-2}} \times \frac{\partial y_{M-2}}{\partial W_i^{(k)}} \\
& \qquad\qquad .... \\
\frac{\partial y_{k+1}}{\partial W_i^{(k)}} &= \frac{\partial y_{k+1}}{\partial y_k} \times \frac{\partial y_k}{\partial W_i^{(k)}}
\end{aligned}
\tag{2.4}
$$

In 2.4, y_r is the output of r-th layer $(out_i = y_{N-1})$. This is called *background error propagation* or *backpropagation*, where the loss error $\frac{\partial L}{\partial y_{N-1}}$ is propagated in the opposite direction of inference networks. In the backward network, the intermediate results of layer t is $e_t = \frac{\partial L}{\partial y_{M-1-t}}$ and the parameters in the backward network are $\frac{\partial y_t}{\partial y_{t-1}}$. We have,

$$
\begin{aligned}
\frac{\partial L}{\partial y_t} &= \frac{\partial y_t}{\partial y_{t-1}} \times \frac{\partial y_L}{\partial y_{t-1}} \\
e_t &= \frac{\partial y_t}{\partial y_{t-1}} + e_{t+1}
\end{aligned}
\tag{2.5}
$$

Therefore, one can rewrite gradient calculation in equation 2.4 as follows.

$$\begin{aligned} e_0 &= \frac{\partial L}{\partial y_{M-1}} \\ e_1 &= \frac{\partial y_{M-1}}{\partial y_{M-2}} \times e_0 \\ &\dots. \\ e_k &= \frac{\partial y_{k-1}}{\partial y_{k-2}} \times e_{k-1} \\ \frac{\partial L}{\partial W_i^{(K)}} &= e_k \times \frac{\partial y_{k-1}}{\partial W_i^{(K)}} \end{aligned} \tag{2.6}$$

$\frac{\partial y_{k-1}}{\partial W_i^{(K)}}$ depends on the input of Layers k $(i.e., y_{k-1})$. In other words, in the process of weight update both $y_i s$ and $e_i s$ are needed.

In the following part of this chapter we discuss the functionality of some of the most popular layers.

2.1.3 Neural Network Layers

In this section we review some of the most popular layers deployed in deep learning architecture.

Fully-connection layer(FC)

This is the most used layer in the history of neural networks. In this layer, every output neuron is the weighted sum of every input neuron 2.7. The layer is illustrated as a bipartite graph with one side representing input neurons while the other sides are output neurons. In between any pair of input neuron, N_i, and output neuron, M_j, there is an edge labeled with the weight $w_{i,j}$. This layer can also be represented as a matrix by vector multiplication $WN = M$, where W, N, and M are the weight matrix, the vector of input neuron values, and the vector of output neuron values, respectively. we have

$$M_j = \sum_{i=0}^{n-1} W_{i,j} \times N_i \tag{2.7}$$

where n is the number of element input neurons. Similarly, we define m as the number of output neurons. With the above notation, we can now derive the backpropagation rules for FC layer.

$$\begin{aligned}\frac{L}{\partial N_i} &= \sum_{j=0}^{m-1} \frac{\partial L}{\partial M_j} \times \frac{\partial M_j}{\partial N_j} \\ \frac{L}{\partial N_i} &= \sum_{j=0}^{m-1} \frac{\partial L}{\partial M_j} \times W_{i,j}\end{aligned} \tag{2.8}$$

if $e_{in} = \left[\frac{L}{\partial N_i}\right]_{0 \leq i < n}$ and $e - out = \left[\frac{L}{\partial M_i}\right]_{0 \leq i < m}$ are the input and output error vectors, we can write:

$$e_{in} = W^T \times e_{out} \tag{2.9}$$

where T is the matrix transpose operation. In addition to the error propagation, we have to calculate the gradient of each weight with respect to the output layer.

$$\begin{aligned}\frac{\partial L}{\partial W_{i,j}} &= \frac{\partial L}{\partial M_j} \times \frac{\partial M_j}{\partial W_{i,j}} \\ \frac{\partial L}{\partial W_{i,j}} &= \sum_{t=0}^{m-1} \frac{\partial L}{\partial M_t} \times \frac{\partial M_t}{\partial W_{i,j}} \\ \frac{\partial L}{\partial W_{i,j}} &= \frac{\partial L}{\partial M_j} \times N_i\end{aligned} \tag{2.10}$$

As shown in equation 2.10, the gradient for the FC layer depends on the inputs and the propagated error in the output. FC layeers requires $m \times n$ parameters, $m \times n$ multiplications and $m \times n$ additions.

Convolution Layer

In the FC layer, all input neurons have influence on all the output neurons, which causes two problems: 1. the FC layer cannot preserve features that depend on the spatial locality and 2. the number of parameters and operations increase superlinearly. Convolution Layer has been proposed to address these two weaknesses. In a convolution layer, input and output neurons are organized in an array of *channels*, each of which are 2D arrays of input neurons. By this organization, the input and output are consideredas 3D arrays. For example, in image classification, input image to the neural network is considered as 3 channels of images, one for red color, one blue color, and one for green color. In general, we assume the input has N_i input channels of $N_x^{in} \times N_y^{in}$ and the outputs consists of N_o output channels of $N_x^{out} \times N_y^{out}$. In our notation, we call input channel i and output channel j as $ch_i^{in}$and ch_j^{out}, respectively. The parameters are organised as 4D arrays: an $N_i \times N_o$ arrays of *kernels* $K_{i,j}$, where kernel is a $Ky \times K_y$ array of weights. Figure 2.1 depicts the general convolution layer organization. Using the above notation, one can write the convolution layer function as follows.

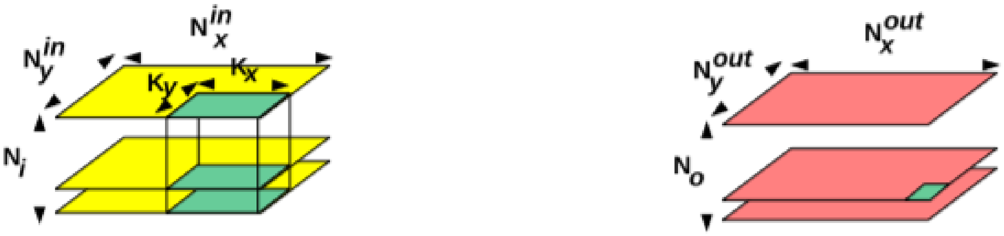


Figure 2.1. The organization of a CNN layer

$$ch_j^{out} = \sum_{r=0}^{N_{in}-1} ch_r^{in} \otimes K_{r,j} \tag{2.11}$$

In 2.10, the summation on channels is element-wise summation. The operation $\otimes$ is 2D convolution operation with two 2D arrays and generates one 2D array outputs. In general, 2D convolution is performed using the following equation.

$$\begin{aligned} B &= A \otimes K \\ B[i][j] &= \sum_{t=0}^{K_y-1} \sum_{r=0}^{K_x-1} A[i+s_x+r]+r[j+s_y+t] \times K[r][s] \\ for s_x = s_y = 1 \Rightarrow B[i][j] &= \sum_{t=0}^{K_y-1} \sum_{r=0}^{K_x-1} A[i+r][j+t] \times K[r][s] \end{aligned} \tag{2.12}$$

One can describe this operation as the kernel K rolling over 2D array A in multiple steps.

In each step, one output entry is calculated by performing inner product of K with the part of A covered by K.2.12, s_x and s_y are the strides in x and y dimensions. Notice that a convolution layer is the general case of the FC layer, where input neurons are replaced with 2D channels, weights are replaced with 2D kernels, and the product of an input and a weight are replaced with 2D convolution operation. If channel and kernels are 1×1, we end up with an FC layer.

One advantage of this interpretation is that we can leverage the FC equations for backpropagation. However, we still need to understand how 2D convolution operations impact error propagation. To this end, we first looked into a case where we have one input $A_{n_1 \times n_2}$ and output channels $B_{m_1 \times m_2}$ with kernel $K_{x \times y}$.If we know the impact of error propagation in this case, we can extend it to convolution layers with more input and/or output channels, with the help of equations developed for the FC layer.

Assuming $s_x = s_y = 1$, we have:

$$\begin{aligned} \frac{\partial L}{\partial A[m][n]} &= \sum_r \sum_t \frac{\partial L}{\partial B[r][t]} \times \frac{\partial B[r][t]}{\partial A[m][n]} \\ \frac{\partial L}{[m][n]} &= \sum_{i=0}^{x-1} \sum_{j=0}^{y-1} \frac{\partial L}{\partial B[m-i][n-j]} \times \frac{\partial B[m-i][n-j]}{\partial A[m][n]} \\ \frac{\partial L}{[m][n]} &= \sum_{i=0}^{x-1} \sum_{j=0}^{y-1} \frac{\partial L}{\partial B[m-i][n-j]} \times K[i][j] \end{aligned} \tag{2.13}$$

If $e_A = [\frac{\partial L}{\partial A[i][j]}]$ and $e_B = [\frac{\partial L}{\partial B[i][j]}]$ are the error maps for the input and output channel, then one can write:

$$
\begin{aligned}
\frac{\partial L}{\partial A[m][n]} &= \sum_{i=0}^{x-1}\sum_{j=0}^{y-1} \frac{\partial L}{\partial B[m-i][n-j]} \times K[i][j] \\
e_A[m][n] &= \sum_{i=0}^{x-1}\sum_{j=0}^{y-1} e_B[m-i][n-i] \times K[i][j] \\
\text{define } m' = (m-x+1) &\text{ and } n' = (n-y+1) \\
e_A[m][n] = \sum_{i=0}^{x-1}\sum_{j=0}^{y-1} & e_B[m'+(x-1-i)][n'+(y-1-j)] \times K[i][j] \\
\text{define } i' = (x-1-i) &\text{ and } j' = (y-1-j) \\
e_A[m][n] = \sum_{i=0}^{x-1}\sum_{j=0}^{y-1} & e_B[m'+i'][n'+j'] \times K[x-1-i'][y-1-j']
\end{aligned}
\tag{2.14}
$$

By changing the variables we can rewrite the equation as:

$$
\begin{aligned}
e_B^{pad}[m][n] = e[m-x+1][n-y+1] = e[m'] & \mathit{if} m \geq x-1, n \geq y-1 \\
\text{otherwise } & \Rightarrow e_B^{pad}[m][n] = 0 \\
\text{Also define } & K'[i][j] = [x-1-j][y-1-j] \\
\text{define } i' = (x-1-i) & \text{ and } j' = (y-1-j) \\
e_A[m][n] = \sum_{i'=0}^{x-1}\sum_{j'=0}^{y-1} & e_B[m'+i'][n'+j'] \times K[x-1-i'][y-1-j'] \\
e_A[m][n] = \sum_{i=0}^{x-1}\sum_{j=0}^{y-1} & e_B^{pad}[m+i][n+i] \times K'[i][j] \\
e_A = & e_B^{pad} \otimes K'
\end{aligned}
\tag{2.15}
$$

In other words, error map in the input channel is the convolution of output channel error maps that has been padded with zeros (i.e., e_B^{pad}) with the rotated version of the original kernel (i.e., K').

In general for N_i input channels and N_o output channels, we have:

$$e_{ch_i^{in}} = \sum_{j=0}^{N_0-1} e^{pad}_{ch_i^{out}} \otimes K'_{i,j} \tag{2.16}$$

Similarly, the weight update array for kernel $K_{i,j}$ is calculated as follows:

$$\frac{\partial L}{\partial K_{i,j}} = e^{pad}_{ch_i^{out}} \otimes ch_i^{in} \tag{2.17}$$

In general, we can state that both forward and backward operations are convolutional operations. The number of parameters in this layer is $N_o \times N_i \times K_x \times K_y$ and the number of operations for additions and multiplications $N_o \times N_x^{out} \times N_y^{out} \times (N_i \times K_x \times K_y)$.

Pooling Layer

As we mentioned, the number of operations in the convolution layer depends on the size of channels. A pooling layer is proposed to down-sample output channels of the convolution layers. A pooling layer is applied per channel. Therefore, it preserves the number of channels in the input. However, the output channels have smaller dimensions. There are two common types of pooling layers, average pooling and max pooling. Average pooling is a 2D convolution operation of Kernel $K_{K_x \times K_y}$ with all its weights equal to $\frac{1}{K_x \times K_y}$. Max pooling, on the other hand, is a 2D convolution with Kernel $1_{K_x \times K_y}$ that uses max operation instead of addition. It is also worth noting that the strides in pooling layers are typically greater than one to reduce the dimensions of the resulting output channels.

Since the average pooling is essentially 2D convolution, we can apply 2.15 to calculate its error maps. For max pooling, error in the output is just propagated to the input with the maximum values. When a pooling layer with kernel of size $K_x \times K_y$ kernel and strides of size s_x and s_y is applied to N_i input channels of size $N_x \times N_y$, $\frac{N_x \times N_y \times K_x \times K_y}{s_x \times s_y}$operations are required.

NonLinear Layers

The secret ingredient in DNNs is nonlinearity. Without nonlinear layers, DNNs are simply a polynomial function of input values. There are three types of nonlinear layers; sigmoid, tanh, and ReLU. These functions are represented by the following equations:

$$\begin{aligned}\sigma(x) &= \frac{1}{1+e^{-x}} \\ tanh(x) &= 2\sigma(2x)-1 \\ ReLU(x) &= max(0,x)\end{aligned} \tag{2.18}$$

Many recent DNNs have adopted ReLU due to its simplicity and high accuracy. However, sigmoid and tanh are still used in LSTMs (Long Short Term Memory). Additionally, some work suggests to approximate the exponential operator in sigmoid and tanh with piece-wise linear functions. Although sigmoid and tanh have exponential operators, they are simply differentiable based on the forward path values.

$$\begin{aligned}\frac{\partial\sigma(x)}{\partial x} &= \sigma(x)(1-\sigma(x)) \\ \frac{\partial tanh(x)}{\partial x} &= (1+tanh(x))\times(1-tanh(x)) \\ \frac{\partial ReLU(x)}{\partial x} &= \left\{\begin{matrix} 1 & if & x\geq 0 \\ 0 & if & x<0 \end{matrix}\right\}\end{aligned} \tag{2.19}$$

2.1.4 Summary

In this section, we reviewed common layers used in state-of-the-art DNNs. We showed that computation intensive layers such as convolutions and FC have the same type of operations in both forward and backward paths.

2.2 Dark Silicon and rise of Domain Specific architecture

2.2.1 Introduction

For the past three decades Moore's Law, coupled with Dennard scaling, has resulted in commensurate exponential performance increases. Shift to multi cores proportionately increased performance. However, with failure of Dennard's scaling and thus slowed supply voltage scaling results in hindrance of core count scaling. As future designs are power limited, higher core counts must provide performance gains despite the worsening energy. Since the energy efficiency of devices is not scaling along with integration capacity, and since few applications have parallelism levels that can efficiently use a 100-core or 1000-core chip, it is critical to understand how good multicore performance will be in the long term. Such a study must consider devices, core microarchitectures, chip organizations, characteristics, benchmark, applying area and power limits at each technology node. In this paper all the parameters mentioned are considered to project upper-bound performance achievable through multicore scaling, and measuring the effects of non-ideal device scaling, including the percentage of dark silicon on future multicore chips.[9]

Three models are combined to project the performance and the fraction of dark silicon on fixed size and fixed power chips. Device scaling model, Core scaling model, Multi core scaling model, all are combined to project the amount of parallelism achievable. Device×core scaling models combined predict Pareto frontiers at future technology nodes and predicts that any performance improvements for future cores will come only at the cost of area or power as defined by these curves. Device×core×multicore scaling models combined predict maximum multicore speedups for future technology nodes while enforcing area, power, and benchmark constraints. On evaluating the models its predicted that over 5 technology generations only 7.9X speedup was possible using ITRS scaling as power limitation curtails the usable chip fraction. So, radical microarchitectural innovations are necessary to alter the power/performance Pareto frontier to deliver speed-ups commensurate with Moore's Law. As discussed above the figure 2.2 illustrates

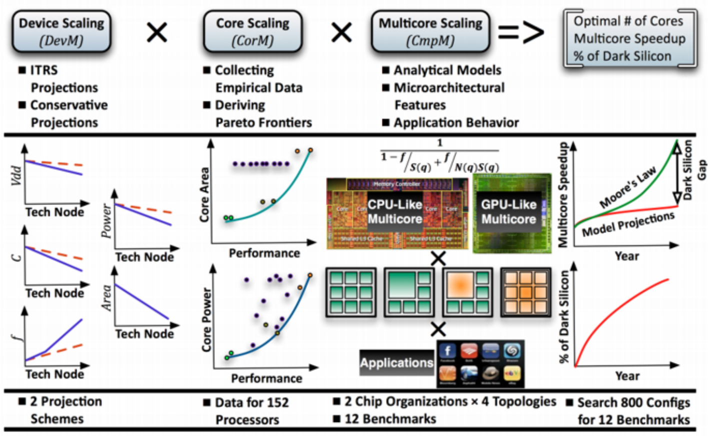


Figure 2.2. Overview of the models and the methodology

how models and empirical measurements combine to project multicore performance and chip utilizations. Multicore scaling model considers two classes: multi core CPUs and many thread GPUs which are extremes in the thread per core spectrum. Many variations of the multicore system are considered like symmetric, asymmetric, dynamic, and composed multicores.Using these configurations, the paper describes analytic model that provides system level performance. The model considers application behavior, memory access patterns, thread level parallelism and micro architecture features like cache size and memory bandwidth. PARSEC which has a set of highly parallel applications is being used as workload. In the next section all the three models are described in detail.

2.2.2 Prediction Models

Device model

Device scaling model was built to provide area, power, and frequency scaling factor using ITRS projections. Two different scaling schemes based on ITRS 2010 and conservative scaling by Borkar are used to device this scaling model. The parameters used for calculating power and performance scaling factors are mentioned in the table 2.1 below. For ITRS scaling, frequency is scaled linearly with respect to FO4 inverter delay. The power scaling is computed using predicted values for $P = aCV^2f$.

Table 2.1. Scaling factors for ITRS and Conservative projections.

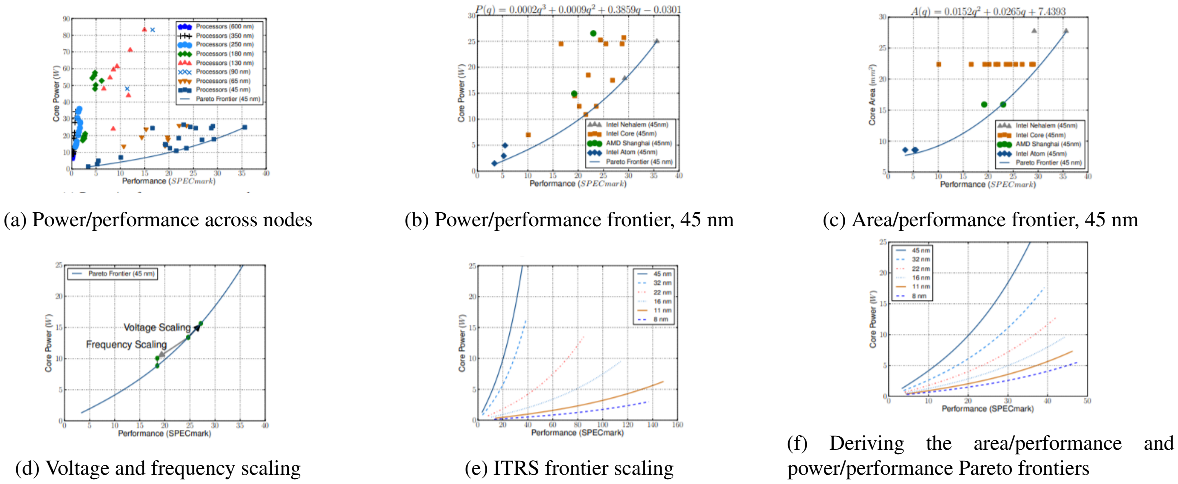
	Year	Tech Node (nm)	Frequency Scaling Factor (/45nm)	Vdd Scaling Factor (/45mm)	Capacitance Scaling Factor (/45nm)	Power Scaling Factor (/45nm)
ITRS	2010	45*	1.00	1.00	1.00	1.00
	2012	32*	1.09	0.93	0.7	0.66
	2015	22 †	2.38	0.84	0.33	0.54
	2018	16 †	3.12	0.75	0.21	0.38
	2021	11 †	4.17	0.68	0.13	0.25
	2024	8 †	3.85	0.62	0.08	0.12
31% frequency increase and 35% power reduction per node						
Conservative	2008	45	1.00	1.00	1.00	1.00
	2010	32	1.10	0.93	0.75	0.71
	2012	22	1.19	0.88	0.56	0.52
	2014	16	1.25	0.86	0.42	0.39
	2016	11	1.30	0.84	0.32	0.29
	2018	8	1.34	0.84	0.24	0.22
6% frequency increase and 23% power reduction per node						

*: Extended Planar Bulk Transistors, †:Multi-Gate Transistors

Core model

Core model provides two functions for A(q) and P(q) , which is area/performance and power/performance tradeoff pareto frontiers, where q is single thread performance of core. The power and area are projected using these Pareto frontiers to future technology nodes using the device scaling model. Pollack's rule has been used to denote the tradeoff between transistor count and performance which considers power to be only area dependent constraint. However, power is dependent on area, supply voltage and frequency. But as Voltage and frequency are not scaling at similar rates, Pollack's rule is insufficient and power, area should be decoupled into independent constraints.

The below figure 2.3 shows the power/performance single-core design space at various nodes. The optimal points show the pareto frontier. The figure 2.3 shows the model design/ cubic pareto frontier P(q) at 45nm node but for different processors. The area/performance quadratic relation was derived in figure2.3. The focus of this work is to study the impact of power constraints on logic scaling, so we derive the pareto frontier using chip power budget (TDP) allocated to each core. Now to compute the power budget of multi core the total power is divided by the number of cores and 30% of it is assigned to the leakage power. To derive pareto frontier at 45nm we fit cubic and quadratic polynomials P(q) and A(q) along the edges of respective design spaces. We used the least square regression method for curve fitting such that the frontiers enclose all design points. The points along the Pareto frontier are used as the search space for determining the best core configuration by the multicore-scaling model.



(a) Power/performance across nodes

(b) Power/performance frontier, 45 nm

(c) Area/performance frontier, 45 nm

(d) Voltage and frequency scaling

(e) ITRS frontier scaling

(f) Deriving the area/performance and power/performance Pareto frontiers

Figure 2.3. Deriving the area/performance and power/performance Pareto frontiers

Below graphs were extrapolated assuming optimal voltage and frequency settings. At fixed Voltage, scaling down frequency, results in a power/performance point inside of the optimal Pareto curve, while scaling voltage up results in a different power-performance point along the frontier. If an application dissipates less power, the voltage and frequency scaling will be utilized to achieve maximum performance with the minimum power increase. This is possible as voltage and frequency scaling changes in a Pareto optimal fashion.

Multicore model

The model first relies on Pollack's rule and then extended to incorporate power as a primary design constraint, independent of area. As per Amdahl's Law, system speedup is 1/(1–f)+ f/S where f is the parallel portion of code and S is the number of cores.

The table 2.2 illustrates the corollaries for each multicore, where TDP is the chip power budget and die area is the area budget. The q denotes performance of single core. Speedup is

Table 2.2. *$CmpM_U$* equations: corollaries of Amdahl's Law for power-constrained multicores.

Symmetric	$\mathrm{N}_{Sym}(q) = min\left(\frac{DIE_{AREA}}{A(q)}, \frac{TDP}{P(q)}\right)$	$\mathrm{Speedup}_{Sym}(f,q) = \frac{1}{\frac{(1-f)}{S_U(q)} + \frac{f}{N_{Sym}(q)S_U(q)}}$
Asymmetric	$\mathrm{N}_{Asym}(q_L,q_S) = min\left(\frac{DIE_{AREA}-A(q_L)}{A(q_S)}, \frac{TDP-P(q_L)}{P(q_S)}\right)$	$\mathrm{Speedup}_{Asym}(f,q_L,q_S) = \frac{1}{\frac{(1-f)}{S_U(q_L)} + \frac{f}{N_{Asym}(q_L,q_S)S_U(q_S)+S_U(q_L)}}$
Dynamic	$\mathrm{N}_{Dyn}(q_L,q_S) = min\left(\frac{DIE_{AREA}-A(q_L)}{A(q_S)}, \frac{TDP}{P(q_S)}\right)$	$\mathrm{Speedup}_{Dyn}(f,q_L,q_S) = \frac{1}{\frac{(1-f)}{S_U(q_L)} + \frac{f}{N_{Dyn}(q_L,q_S)S_U(q_S)}}$
Composed	$\mathrm{N}_{Composed}(q_L,q_S) = min\left(\frac{DIE_{AREA}}{(1+\tau)A(q_S)}, \frac{TDP-P(q_L)}{P(q_S)}\right)$	$\mathrm{Speedup}_{Composed}(f,q_L,q_S) = \frac{1}{\frac{(1-f)}{S_U(q_L)} + \frac{f}{N_{Composed}(q_L,q_S)S_U(q_S)}}$

measured against a baseline core with performance q Baseline. For symmetric cores, the parallel fraction is distributed across the symmetric cores each of which has speedup over baseline. For asymmetric, the number of small cores is bounded by the power consumption or area of the large core. In Dynamic multicore, if area is the dominant constraint, the number of small cores is bounded by the area of the large core. The area overhead supporting the composed topology is T. Thus, the area of small cores increases by a factor of (1 + T) with no power overhead.

The above provide a strict upper-bound on parallel performance, but do not have the level of detail required to explore microarchitectural features and workload behavior. The *$CmpM_R$* model formulates the performance of a multicore in terms of chip organization, frequency, CPI, cache hierarchy, and memory bandwidth, application behaviors such as the degree of thread-level parallelism, the frequency of load and store instructions, and the cache miss rate. Multi thread performance is calculated in terms of instructions per second by multiplying the number of cores by the core utilization and scaling by the ratio of the processor frequency to *CPI_{exe}*:

$$Perf = min\left(N\frac{freq}{CPI_{exe}}\eta, \frac{BW_{max}}{r_m \times m_{L1} \times b}\right) \tag{2.20}$$

The stalls due to cache access are covered separately in core utilization which the fraction of time that a thread running on the core can keep it busy

$$\eta = min\left(1, \frac{T}{1+t\frac{r_m}{CPI_{exe}}}\right) \tag{2.21}$$

The average time spent waiting for memory accesses is a function of the time to access the caches both L1 and L2, time to visit memory, and the predicted cache miss rate for both L1 and L2:

$$\begin{gathered} t = (1 - m_{L1})\, t_{L1} + m_{L1}\, (1 - m_{L2})\, t_{L2} + m_{L1} m_{L2} t_{mem} \\ m_{L1} = \left(\frac{C_{L1}}{T\beta_{L1}}\right)^{1-\alpha_{L1}} \mathit{and}\; m_{L2} = \left(\frac{C_{L2}}{NT\beta_{L2}}\right)^{1-\alpha_{L2}} \end{gathered} \tag{2.22}$$

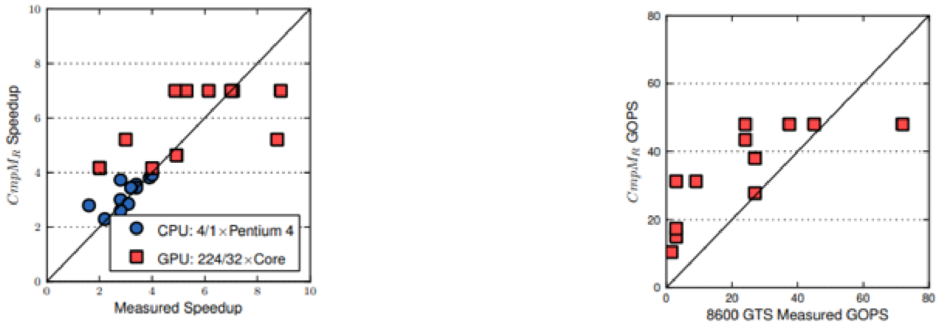
To compute the overall speedup of different multicore topologies using the $CmpM_R$ model, we calculate the baseline multithreaded performance for all benchmarks by providing the $Perf$ equation.To incorporate the Pareto-optimal curves into the $CmpM_R$ model, the SPECmark scores are converted into an estimated CPI_{exe} and core frequency. We assume the core frequency scales linearly with performance. Each application's CPI_{exe} is dependent on its instruction mix and use of hardware optimizations. $CmpM_R$ model is used to generate per benchmark CPI_{exe} estimates for each design point along the Pareto frontiers. With all other model inputs kept constant, we iteratively search for the CPI_{exe} at each processor design point.

Since the performance increase between any two points should be the same using either the SPECmark score or $CmpM_R$ model, we continue in this fashion to estimate a per benchmark CPI_{exe} for each processor design point. We assume CPI_{exe} does not change with technology node, while frequency scales as discussed. This flexible approach allows us to use the SPECmark scores to select processor design points from the Pareto optimal curves and generate reasonable performance model inputs. To characterize an application, the required input parameter models are cache behavior, fraction of instructions that are loads or stores, and fraction of parallel code. To obtain f , the fraction of parallel code, for each benchmark, we fit an Amdahl's Law-based curve to the reported speedups across different numbers of cores from both studies. This fit shows values of parallel code between 0.75 and 0.9999 for individual benchmarks.

We compute serial($Perf_s$) and parallel($Perf_p$) performance with respective cores and their parameters.Assuming performance of single core($Perf_b$). The serial portion is sped up by $S_r, serial = Perf_s / Perf_b$ and parallel is sped up by $S_r, parallel = Perf_p / Perf_b$. Formulation below captures the impact of parallelism on all four topologies:

$$Speedup_R = 1/\left(\frac{1-f}{S_{R,Serial}} + \frac{f}{S_{R,Parallel}}\right) \tag{2.23}$$

A key component of the detailed model is the set of input parameters that model the microarchitecture of the cores. Two styles of core models are used: single-thread and many-thread. For single-thread cores, we assume each core has L1 cache, and chips with only ST cores have an L2 cache that is 30% of the chip area. Many-thread cores have small L1 caches, a thread register file, and no L2 cache. The off-chip bandwidth (BW_max) is assumed to increase linearly as process technology scales down while the memory access time is constant. The model's accuracy is limited by our assumptions which are optimistic. Thus, the model only over-predicts performance, making our speedup projections optimistic. This model allows us to estimate the first-order impact of caching, parallelism, and threading under several key assumptions. It optimistically assumes that the workload is homogeneous, work is infinitely parallel during parallel sections of code, and no thread synchronization, operating system serialization, or swapping overheads occur. We also assume memory accesses never stall due to a previous access. Each of these assumptions could cause the model to overpredict performance, making the model and hence projected speedups optimistic.



(a) Speedup

(b) Performance

Figure 2.4. CmpM$_R$ validation

Figure 2.4, which includes both CPU and GPU data, shows that the model is optimistic.

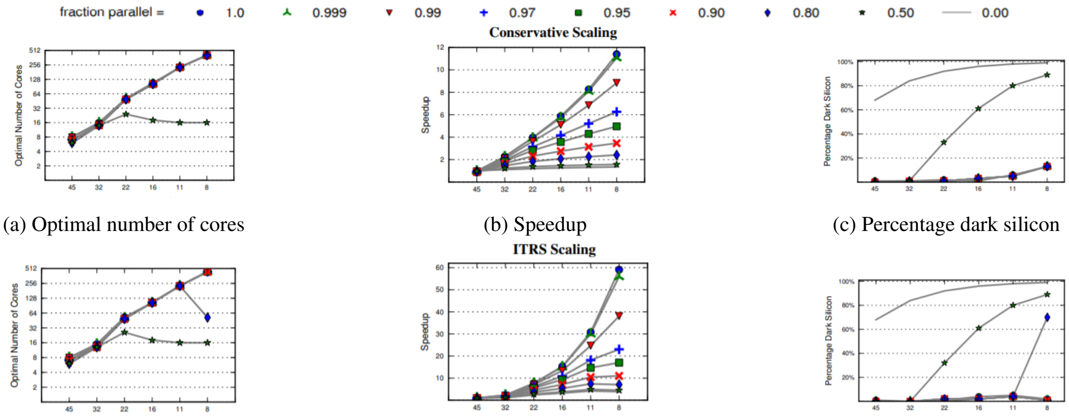
$CmpM_R$ underpredicts speedups for two benchmarks; these speedups are greater than 7. To strongly advance our GPU claim, we also need to prove the model's raw performance projection is accurate or optimistic. As depicted in second graph, the model's GPU performance projection is validated by comparing its output to the results from a real system. Using our model, we find 4 geometric-mean and 12 maximum speed up for PARSEC benchmarks on GPU compared to a quad-core CPU. Our model does not account for specialized compute units, which contribute to the speedup.

DEVICE × CORE × CMP SCALING

The three models are now combined to produce projections for optimal performance, number of cores, and amount of dark silicon. To determine the best core configuration at each technology node, we consider only the processor design points along the $area/performance$ and $power/performance$ Pareto frontiers as they represent the most efficient design points. The following outlines the process for choosing the optimal core configuration for the symmetric topology at a given technology node: The $area/performance$ Pareto frontier is investigated, and all processor design points along the frontier are considered. For each $area/performance$ design point, the multicore is constructed starting with a single core. We add one core per iteration and compute the new speedup and the power consumption using the $power/performance$ Pareto frontier. Speedups are computed using the Amdahl's Law corollaries to obtain an upper-bound or our $CmpM_R$ model for more realistic performance results using the PARSEC benchmarks. The speedup is computed over a CPU. After some number of iterations, the area limit is hit, or power wall is hit, or we start seeing performance degradation. At this point the optimal speedup and the optimal number of cores is found. The fraction of dark silicon can then be computed by subtracting the area occupied by these cores from the total die area allocated to processor cores. The above process is repeated for each technology node using the scaled Pareto frontiers. This exhaustive search is performed separately for Amdahl's Law $CmpM_U$, CPU-like $CmpM_R$, and GPU-like $CmpM_R$ models. We optimistically add cores until either the power or area budget is reached.

2.2.3 Evaluations

We begin the study of future multicore designs with an optimistic upper-bound analysis using the Amdahl's Law multicore-scaling model, $CmpM_U$. Then, to achieve an understanding of speedups for real workloads, we consider the PARSEC benchmarks and examine both CPU-like and GPU-like multicore organizations under the four topologies using our $CmpM_R$ model.



(a) Optimal number of cores (b) Speedup (c) Percentage dark silicon

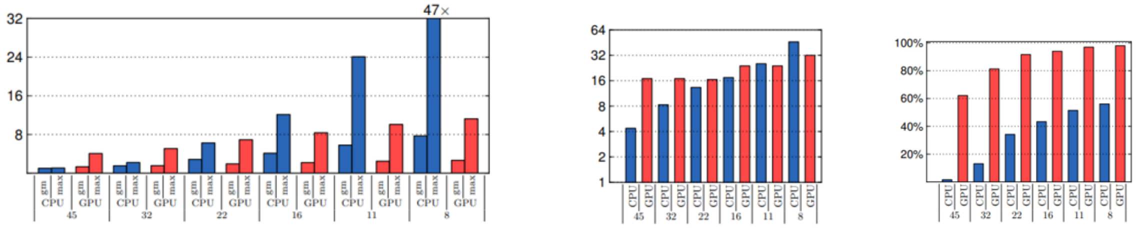
(d) Optimal number of cores (e) Speedup (f) Percentage dark silicon

Figure 2.5. Amdahl's law projections for the dynamic topology. Upperbound of all four topologies (x-axis: technology node).

Above figure 2.5 show the multicore scaling results comprising the optimal number of cores, achievable speedup, and dark silicon fraction under conservative scaling. Below figures show the same results using ITRS scaling. The results are only presented for the dynamic topology, which offers the best speedup levels amongst the four topologies.

The 59 speedup at 8 nm for highly parallel workloads using ITRS predictions, which exceeds the expected 32, is due to the optimistic device scaling projections. We consider scaling of the Intel Core2 Duo T9900 to clarify. At 45 nm, the T9900 has a SPECmark of 23.58, frequency of 3.06 GHz, TDP of 35 W and per-core power of 15.63 W and are of 22.30 mm^2 . With ITRS scaling at 8nm, T9900 will have SPECmark of 90.78, frequency of 11.78 GHz, core power of 1.88 W, and core area of 0.71 mm^2 .

With the 125 W power budget at 8nm, 67 such cores can be integrated. There is consensus that such power efficiency is unlikely. Further, our $CmpM_U$ model assumes that performance scales linearly with frequency. These optimistic device and performance assumptions result in speedups exceeding Moore's Law. Considering PARSEC applications executing on CPU- and GPU-like chips. The study considers all four multicore topologies using the $CmpM_R$ realistic model. This model captures microarchitectural features as well as application behavior. To conduct a fair comparison between different design points, all speedup results are normalized to the performance of a quadcore Nehalem multicore at 45 nm.



(a) Speedup: geomean and best case

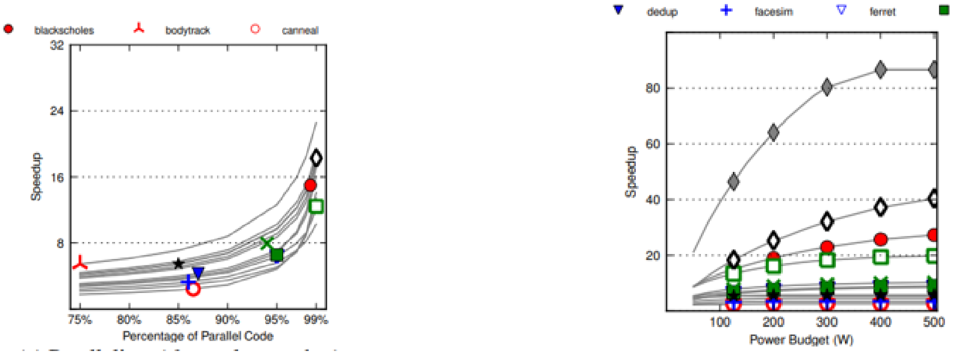
(b) Number of cores: geomean

(c) Percent dark silicon: geomean

Figure 2.6. Speedup and number of cores across technology nodes using symmetric topology and ITRS scaling

Figure 2.6 shows the geometric mean of speedup, best-case speedup, geometric mean of the optimal number of cores, and geometric mean of the percentage dark silicon using optimistic ITRS scaling. The symmetric topology achieves the lower bound on speedups, with speedups that are no more than 10% higher, the dynamic and composed topologies achieve the upper-bound. The results are presented for both CPU like and GPU-like multicore organizations.

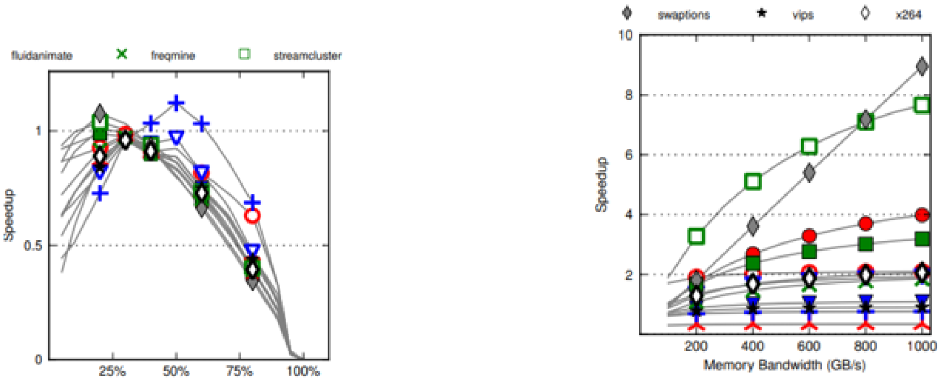
The optimal number of cores projected by our study seems small compared to chips such as the NVIDIA Fermi, which has 512 cores at 45 nm. There are two reasons for this discrepancy. First, in our study we are optimizing for a fixed power budget, whereas with real GPUs the power has been slightly increasing. Second, our study optimizes core count and multicore configuration for general purpose workloads like the PARSEC suite. We assume Fermi is optimized for graphics rendering. When we applied our methodology to a graphics kernel in an asymmetric topology, we obtained higher speedups and an optimal core count at 8 nm, with 8% dark silicon.



(a) Parallelism (f actual at marker)

(b) Power

Figure 2.7. Dark silicon bottleneck relaxation using CPU organization and dynamic topology at 8 nm with ITRS Scaling



(a) L2 size (CPU)

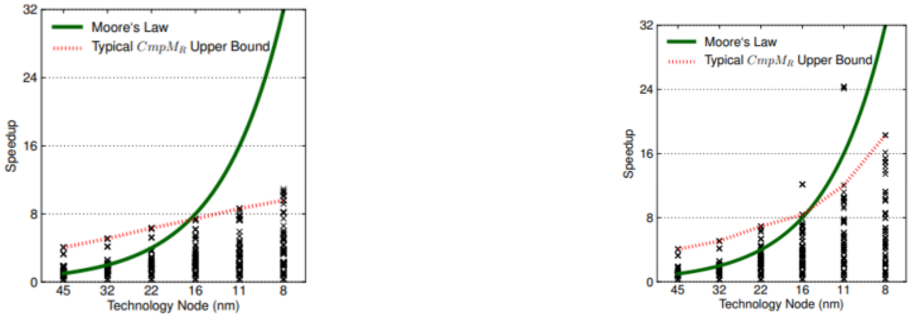
(b) Sensitivity studies of L2 size and memory bandwidth using symmetric topology at 45 nm

Figure 2.8. Deriving the area/performance and power/performance Pareto frontiers

To understand whether parallelism or power is the primary source of dark silicon, we examine our model results with power and parallelism levels alone varying in separate experiments as shown in Figure 2.7 for the 8 nm node. First, we set power to be the only constraint, and vary the level of parallelism in the PARSEC applications from 0.75 to 0.99, assuming programmer effort can somehow realize this. As shown in figure 2.8, which normalizes speedup to a quad-core Nehalem at 45 nm, we see performance improves only slowly as the parallelism level increases, with most benchmarks reaching a speedup of about only 15 at 99% parallelism. The markers show the level of parallelism in their current implementation. If power is the only constraint, typical ITRS scaling speedups would still be limited to 15. With conservative scaling, this best-case speedup is 6.3. We then see what happens if parallelism alone was the constraint by allowing the power budget to vary from 50 W to 500 W in Figure 2.8. Eight of twelve benchmarks show no more than 10X speedup even with practically unlimited power, i.e. parallelism is the primary contributor to dark silicon.

Only four benchmarks have sufficient parallelism to sustain Moore's Law level speedup even hypothetically, but dark silicon due to power limitations constrains what can be realized. Our model allows us to do such studies and shows that only small benefits are possible from such simple changes. We elaborate on two representative studies below. Figure 2.8 for L2 cache area shows the optimal speedup at 45 nm as the amount of a symmetric CPU's chip area devoted to L2 cache varies from 0% to 100%. In this study we ignore any increase in L2 cache power or increase in L2 cache access latency. Across the PARSEC benchmarks, the optimal percentage of chip devoted to cache varies from 20% to 50% depending on benchmark memory access characteristics. Compared to a 30% cache area, using optimal cache area only improves performance by at most 20% across all benchmarks. Figure 2.8 for Memory bandwidth illustrates the sensitivity of PARSEC performance to the available memory bandwidth for symmetric GPU multicores at 45 nm. As the memory bandwidth increases, the speedup improves as the bandwidth can keep more threads fed with data; however, the increases are limited by power and/or parallelism and in 10 out of 12 benchmarks speedups do not increase by more than 2 compared to the baseline, 200 GB/s.

Figure 2.9 below summarizes all the speedup projections in a single scatter plot. For every benchmark at each technology node, eight possible configurations, (CPU, GPU) (symmetric, asymmetric, dynamic, composed) are plotted. The solid curve indicates performance Moore's Law or doubling performance with every technology node. As depicted, due to the power and parallelism limitations, a significant gap exists between what is achievable and what is expected by Moore's Law. Results for ITRS scaling are slightly better but not by much. With conservative scaling a speedup gap of at least 22 exists at the 8 nm technology node compared to Moore's Law. Assuming ITRS scaling, the gap is at least 13 at 8 nm.



(a) Conservative Scaling

(b) ITRS Scaling

Figure 2.9. Speedup across process technology nodes over all organizations and topologies with PARSEC benchmarks

2.2.4 Summary

For decades Dennard scaling projected faster and energy efficient transistors with each technology node. However, with failure of Dennard scaling principle industry shifted towards multicore path, which permitted performance scaling for parallel and multi tasked workloads. But as the benefits of multicore scaling begin to end, a new driver of transistor utility must be found. An essential question is how much more performance can be extracted from the multicore path in the near future. This paper combined technology scaling models, performance models, and empirical results from parallel workloads to answer that question and estimate the remaining performance available from multicore scaling. Using PARSEC benchmarks and ITRS scaling projections, this study predicts best-case average speedup of 7.9 times between now and 2024 at 8 nm. That result translates into a 16% annual performance gain, for highly parallel workloads and if each benchmark has its ideal number and granularity of cores. However, we believe that the ITRS projections are much too optimistic, especially in the challenging sub-22 nanometer environment. The conservative model we use in this paper more closely tracks recent history. Applying these conservative scaling projections, half of that ideal gain vanishes; the path to 8nm in 2018 results in a best-case average 3.7 speedup; approximately 14% per year for highly parallel codes and optimal per-benchmark configurations. The returns will certainly be lower in practice. Currently, the broader computing community is in consensus that we are in "the multicore era.” Given the low performance returns assuming conservative scaling, adding more cores will not provide sufficient benefit to justify continued process scaling. If multicore scaling ceases to be the primary driver of performance gains at 16nm (in 2014) the “multicore era” will have lasted a mere nine years, a short-lived attempt to defeat the inexorable consequences of Dennard scaling’s failure. Clearly, architectures that move well past the Pareto-optimal frontier of energy/performance of today’s designs will be necessary. Given the timeframe of this problem and its scale, radical or even incremental ideas simply cannot be developed along typical academic research and industry product cycles. On the other hand, left to the multicore path, we may hit a “transistor utility economics” wall in as few as three to five years, at which point Moore’s Law may end, creating massive disruptions in our industry. Hitting a wall from one of these two directions appears inevitable. There is a silver lining for architects, however: At that point, the onus will be on computer architects–and computer architects only–to deliver performance and efficiency gains that can work across a wide range of problems. It promises to be an exciting time.

Chapter 3

Related Work

3.1 Introduction

The need for higher speed and efficiency in DNN execution has led to explosion of DNN accelerators. The range of innovative AI hardware-accelerator architectures continues to expand.Over the past several years, both startups and established chip vendors have introduced an impressive new generation of new hardware architectures optimized for machine learning, deep learning, natural language processing, and other AI workloads. Today's AI market has no hardware mono culture equivalent to Intel's x86 CPU, which once dominated the desktop computing space. That's because these new AI-accelerator chip architectures are being adapted for highly specific roles in the burgeoning cloud-to-edge ecosystem, such as computer vision.

The myriad AI chipset architectures on the market reflect the diverse range of machine learning, deep learning, natural language processing, and other AI workloads that range from storage-intensive training to compute-intensive inference and involve varying degrees of device autonomy and person-in-the-loop interactivity. To address the range of workloads that AI chipsets are being used to support, vendors are mixing a wide range of technologies in their product portfolios. In following sections we discuss how AI chips started and we discuss in details the steps taken to enhance the computation as well as memory fetch capabilities of hardware accelerator chips.

3.2 Basic Design

3.2.1 Introduction

Architectures are evolving towards heterogeneous multi cores and realizing the best tradeoff between efficiency and flexibility is important. However, with DNNs and CNNs proving to be best across wide range of applications there is an opportunity to design accelerators with significant scope and high performance and efficiency. Most of the work has been focused on the computation part and memory transfers have been ignored, which might hinder the performance throughput considering Amdahl's law.This paper focuses on basic accelerator design for NN by minimizing memory transfer and improving efficiency. [4].

3.2.2 Architecture

The most natural way to map a neural network onto silicon is to fully layout the neurons and synapses. As shown in figure 3.1 hardware neuron performs the following operations: multiplication of inputs and synapses, addition of all such multiplications, followed by a sigmoid.

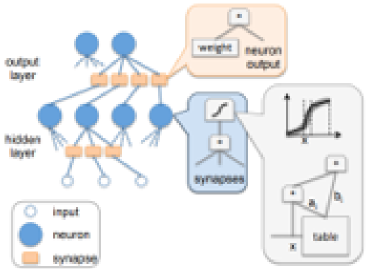


Figure 3.1. Full hardware implementation of neural networks

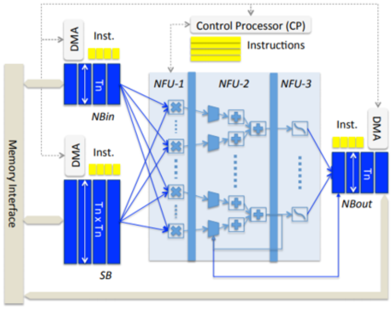


Figure 3.2. Accelerator

The neurons, synapses are implemented as logic circuits and latches or RAMs. This can be done for small networks and energy reduction has been reported. The area, energy and delay grow quadratically with the number of neurons. This full hardware layout is not realistic for large networks as there would be many layers. So, the principle was to timeshare the physical neurons and use the on-chip RAM to store synapses and intermediate neurons values of hidden layers. Large scale neural networks are implemented in a different way to accommodate for change in scale as shown in figure 3.2.

The main components are input, output buffers for input/output neurons, third buffer for synaptic weights connected to a computational block called neural functional unit and a control logic. The architecture can be explained by dividing entire block into computation, storage and control code. NFU is used to reflect the decomposition of a layer into computational blocks of inputs/synapses and output neurons. The computation of each layer can be decomposed into 2 or 3 stages. For classifier, convolution we have multiplication of synapses and inputs followed by addition and finally sigmoid or other nonlinear functions. For pooling it is average or max. We can pipeline all 2 or 3 operations, but the pipeline must be staggered. the first or first two stages operate as normal pipeline stages, but the third stage is only active after all additions have been performed. Sigmoid function can be implemented using piece wise linear interpolation $f = ax + b$. In terms of hardware, it corresponds to two 16x1 16-bit multiplexers for segment boundary selection, one 16-bit multiplier (16-bit output) and one 16-bit adder to perform the interpolation. The 16-segment coefficients (a_i, b_i) are stored in a small RAM and changing these value changes the function implemented. 16-bit fixed-point arithmetic operators instead of word size floating point operator is used as small size have almost no impact on accuracy. Also, the arithmetic operators are truncated using a standard n-bit truncated multiplier.

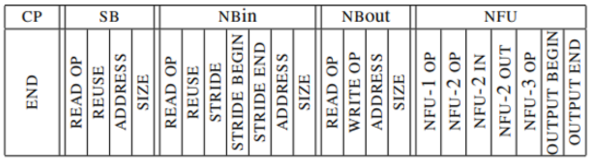
The different storage structures of the accelerator can be construed as modified buffers of scratchpads. While caches are good, they have access overheads like tag checks, associativity, line size and cache conflicts. However, scratchpad in a dedicated accelerator realizes efficient storage, and easy exploitation of locality because only a few algorithms must be manually adapted. Storage is split into three structures: input, output and synapse buffers. The first benefit of splitting is to tailor the SRAM to appropriate read and write widths. Significant energy penalty is incurred while reading Nb_{in} and NB_{out} ($T_n \times 2$ bytes) from $T_n \times T_n \times 2$ wide bank while doing vice versa would incur time penalty. So, splitting them would improve both time and energy. Second, to avoid conflicts we need a larger associativity to speculatively read values parallelly for faster access. However, this increases the energy costs so split storage with precise knowledge of locality behavior allows to entirely remove data conflicts.

Each buffer is implemented using DMA to exploit spatial locality. DMA requests are issued in the form of instructions and are decoupled from current requests using FIFO. We can preload request to avoid long latencies if there is buffer space. The inputs of all layers are split into chunks which fit in NB_{in}, and they are reused by implementing NB_{in} as a circular buffer. To make memory fetches more efficient we introduce a mapping function in NBin which has the effect of locally transposing loops k_y, k_x and loop i so that data is loaded along loop i, but it is stored in NBin and thus sent to NFU along loops k_y, k_x first.

Partial sums are stored in dedicated buffer to avoid exit and reloading of them for each entry of Nb_{in} buffer. NB_{out} is used as a temporary storage buffer for the partial sums while reusing input neurons as NB_{out} is idle as along as all the input neurons have not been integrated.

A layer execution is broken down into set of instructions. The instructions are stored in SRAM associated with control processor which drives the execution of DMAs of three buffers and NFU.

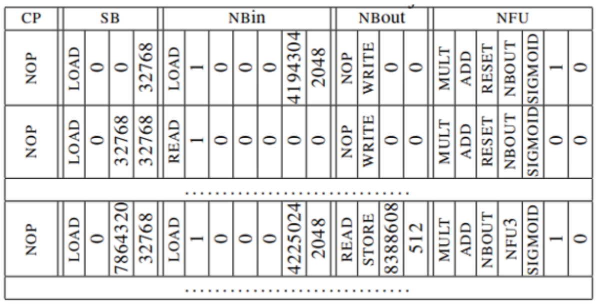
Table 3.1. Control instruction format



CP	SB				NBin							NBout				NFU						
END	READ OP	REUSE	ADDRESS	SIZE	READ OP	REUSE	STRIDE	STRIDE BEGIN	STRIDE END	ADDRESS	SIZE	READ OP	WRITE OP	ADDRESS	SIZE	NFU-1 OP	NFU-2 OP	NFU-2 IN	NFU-2 OUT	NFU-3 OP	OUTPUT BEGIN	OUTPUT END

Every instruction has five slots CP, three buffers and NFU as shown in table 3.1. There are mainly three type of instructions as shown in table 3.2. The first instruction is load, to fetch data from memory. the next instruction is a read, because these input neurons are rotated in the buffer for the next chunk of T_n neurons. As discussed output of NFU-2. For the first (and next) instruction is NB_{out}, i.e., the partial output neurons sums are rotated to NB_{out}. Finally, when the

Table 3.2. Subset of classifier/perceptron code (N_i = 8192, N_o = 256, T_n = 16, 64-entry buffers).



CP	SB				NBin							NBout				NFU						
NOP	LOAD	0	0	32768	LOAD	1	0	0	0	4194304	2048	NOP	WRITE	0	0	MULT	ADD	RESET	NBOUT	SIGMOID	1	0
NOP	LOAD	0	32768	32768	READ	1	0	0	0	0	0	NOP	WRITE	0	0	MULT	ADD	RESET	NBOUT	SIGMOID	0	0
....																						
NOP	LOAD	0	7864320	32768	LOAD	1	0	0	0	4225024	2048	READ	STORE	8388608	512	MULT	ADD	NBOUT	NFU3	SIGMOID	1	0
....																						

last chunk of input neurons is sent, the DMA of NB_{out} is set for STORE.

3.2.3 Evaluation

The accelerator when evaluated performs 62X more 16-bit operations than a SIMD processor as shown in figure 3.3. This could be because of preloading and reuse in Nb_{in} and SB buffers. The accelerator is slow for POOL layer due to the small size of input and output feature maps. Also control and scheduling helped in preventing lost cycles by optimizing pipeline stages. The use of 16 neurons with 16 synapses each makes the accelerator operates at 452 GOP/s. The area and power are dominated by the buffers ($NB_{in}/NB_{out}/SB$) at respectively 56% and 60%, with the NFU being a close second at 28% and 27%. The percentage of the total cell power is 59.47%, but the routing network accounts for a significant share of the total power at 38.77%. At 65nm, due to the high toggle rate of the accelerator, the leakage power is almost negligible at 1.73%.

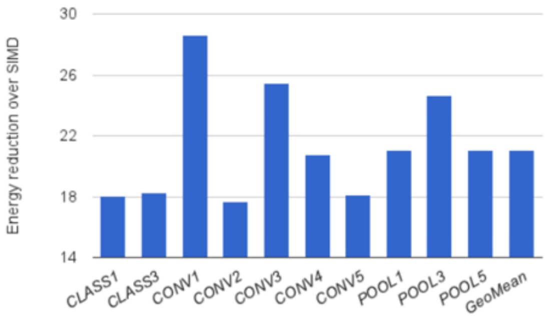


Figure 3.3. Energy reduction of accelerator over SIMD.

We see more energy reduction for layers where computation is around 85%. However, for small reduction the layers are mostly due to memory access. By Amdahl's law memory access dominates the reduction in speed up achieved.

3.2.4 Summary

The focus was to design accelerator for larger scope Machine learning algorithms like CNNs and DNNs. By carefully exploiting the locality properties of intermediate layers, and by introducing storage structures custom designed to take advantage of these properties, we show that it is possible to design a machine-learning accelerator capable of high performance in a very small area footprint. Memory access is the major overhead, however speedup of 117.87X and an energy reduction of 21.08X over a 128-bit 2GHz SIMD core with a normal cache hierarchy is achieved.

3.3 Dataflow Organisation

3.3.1 Introduction

Although basic architecture discussed in previous section can leverage highly parallel compute paradigms, throughput may not scale due to bandwidth requirement, and the energy consumption remains high as data movement can be more expensive than computation. To achieve energy efficiency, we need to develop dataflows that support parallel processing with minimal data movement. The differences in data movement energy cost based on where the data is stored also needs to be accounted for.In this section we discuss the taxonomy that classifies existing CNN dataflows. Finally A novel dataflow called the Row Stationary, that maximizes energy efficiency for CNN acceleration is proposed which exploits all types of data reuse and considers energy cost of data movement at different levels of the storage hierarchy.[13]

3.3.2 Architecture

Spatial architectures are a class of accelerators that can exploit high compute parallelism using direct communication between an array of processing engines (PEs). They support different algorithms which are mapped onto PEs using specialized dataflow and can leverage efficient data sharing across. They support dataflows that exhibit producer consumer relation. SAs come in two flavors: coarse-grained SAs that consist of tiled arrays of ALU-style PEs connected via on-chip networks, and fine-grained SAs that are in the form of an FPGA. Coarse-grained SAs are popular as first, the operations in a CNN layer are uniform and exhibit high parallelism, which can be computed quite naturally with parallel ALU style PEs and Second, direct inter-PE communication can be used very effectively for passing partial sums to achieve spatially distributed accumulation, or sharing the same input data for parallel computation without incurring higher energy data transfers.

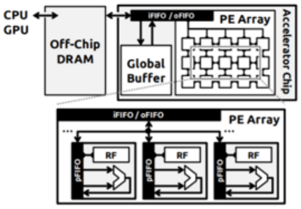


Figure 3.4. Block diagram of a CNN accelerator system consisting of a spatial architecture accelerator and an off-chip DRAM.

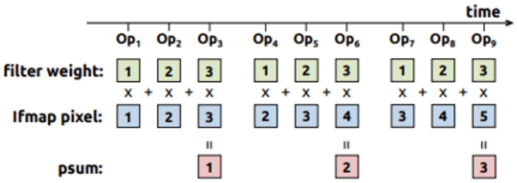


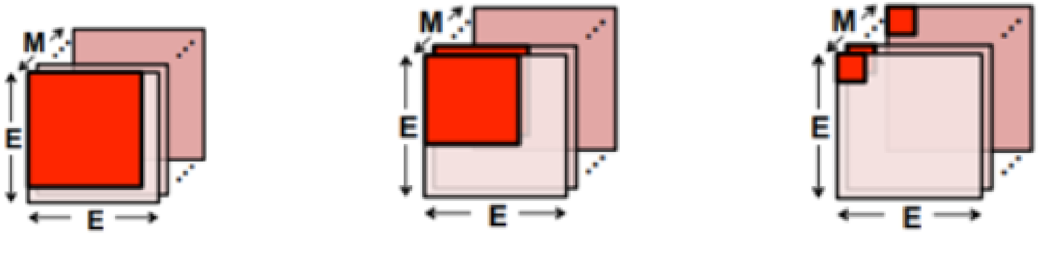
Figure 3.5. Processing of an 1D convolution primitive in the PE. In this example, R=3 and H=5.

The challenge in design lies in the exact mapping of the CNN dataflow to the SA since it has a strong implication on the resulting throughput and energy efficiency. Figure 3.4 shows the high-level block diagram of the accelerator system that is used in this paper for CNN processing. It consists of a SA accelerator and off-chip DRAM. The inputs can be off-loaded from the CPU or GPU to DRAM and processed by the accelerator. The outputs are then written back to DRAM and further interpreted by the main processor. The SA accelerator is primarily composed of a global buffer and an array of PEs. The DRAM, global buffer and PE array communicate with each other through FIFOs. The global buffer can be used to exploit input data reuse and hide DRAM access latency, or for the storage of intermediate data. The PEs are connected via a network on chip, and the NoC design depends on the dataflow used. The PE includes an ALU data path, which can do multiply-and-accumulate and addition, a register file as a local scratchpad, and a PE FIFO used to control the traffic going in and out of the ALU. Overall, the system provides four levels of storage hierarchy for data accesses, including DRAM, global buffer, inter-PE, and RF with different costs for each level.

3.3.3 Existing Dataflows

Based on data handling characteristics, a taxonomy of existing CNN dataflow is presented. In weight stationary dataflow, each filter weight remains stationary in the RF to maximize convolutional reuse and filter reuse. Once a weight is fetched from DRAM to the RF of a PE, the PE runs through all NE^2 operations that use the same filter weight. $R \times R$ weights from the same filter are mapped to PEs and remain stationary while the pixels in an ifmap are broadcasted sequentially. The psums generated by each PE are further accumulated spatially across these PEs. The RF is used to store the stationary filter weights. Due to the operation scheduling that maximally reuses stationary weights, psums are not always immediately reducible, and will be temporarily stored to the global buffer. If the buffer is not large enough, the number of psums that are generated together must be limited.

For, output stationary dataflow, the accumulation of each ofmap pixel stays stationary in a PE. The psums are stored in the same RF for accumulation to minimize the psum accumulation cost. This type of dataflow uses the space of the PE array to process a region of the 4D ofmap at a time and regions are selected in two ways possible: multiple ofmap channels vs. single ofmap channels, and multiple ofmap-plane pixels vs. single ofmap-plane pixels. This creates three practical OS dataflow subcategories: SOC-MOP, MOC-MOP, and MOC-SOP. SOC-MOP is used mainly for CONV layers and focuses on processing a single plane of ofmap at a time. It further maximizes convolutional reuse in addition to psum accumulation. MOC-MOP processes multiple ofmap planes with multiple pixels in the same plane at a time. By doing so, it tries to further exploit both convolutional reuse and ifmap reuse. MOC-SOP is used mainly for FC layers, since it processes multiple ofmap channels but with only one pixel in a channel at a time. It focuses on further exploiting ifmap reuse.



(a) SOC-MOP (b) MOC-MOP (c) MOC-SOP

Figure 3.6. Comparison of the three different OS dataflow variants. The red blocks depict the ofmap region that the OS dataflow variants process at once.

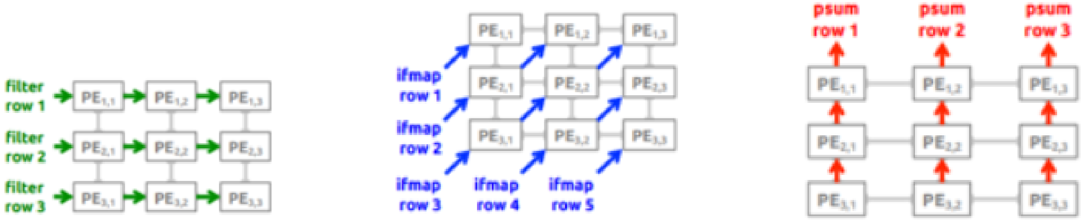
The difference between the three OS dataflows is illustrated in Figure 3.6. All OS dataflows use the RF for psum storage to achieve stationary accumulation. In addition, SOC-MOP and MOC-MOP require additional RF storage for ifmap buffering to exploit convolutional reuse within the PE array.

No Local Reuse (NLR) dataflow has two major characteristics: it does not exploit data reuse at the RF level, and it uses inter-PE communication for ifmap reuse and psum accumulation. NLR divides the PE array into groups of PEs. PEs within the same group read the same ifmap pixel but with different filter weights from the same input channel. Different PE groups read ifmap pixels and filter weights from different input channels. The generated psums are accumulated across PE groups and do not need RF storage so there is lot of global memory to store psums.

3.3.4 Energy-Efficient Dataflow

It would be desirable if a dataflow could optimize for all types of data movement energy costs. The Row stationary dataflow breaks the high-dimensional convolution down into 1D convolution primitives that can run in parallel, each primitive operates on one row of filter weights and one row of ifmap pixels and generates one row of psums. Psums from different primitives are further accumulated together to generate the ofmap pixels. The inputs to the 1D convolution come from the storage hierarchy. Each primitive is mapped to one PE for processing; therefore, the computation of each row pair stays stationary in the PE, which creates convolutional reuse of filter weights and ifmap pixels at the RF level. An example of this sliding window processing is shown below in figure 3.5.

However, due to large size, the exact mapping of all primitives to the PE array is not simple and will affect energy efficiency. The primitive mapping is separated into two steps: logical mapping and physical mapping. The logical mapping first deploys the primitives into a logical PE array, which has the same size as the number of 1D convolution primitives and is usually much larger than the physical PE array. The physical mapping then folds the logical PE array, so it fits into the physical PE array. Folding implies serializing the computation and is determined by the amount of on-chip storage. The two mapping steps happen statically prior to runtime, so no on-line computation is required.



(a) rows of filter weight are reused across PEs horizontally.

(b) rows of ifmap pixel are reused across PEs diagonally.

(c) rows of psum are accumulated across PEs vertically.

Figure 3.7. The dataflow in a logical PE set to process a 2D convolution. In this example, R= 3 and H= 5.

Each 1D primitive is first mapped to one logical PE in the logical PE array. Since there is considerable spatial locality between the PEs that compute a 2D convolution in the logical PE array, we group them together as a logical PE set. Figure 3.7 shows a logical PE set, where each filter row and ifmap row are horizontally and diagonally reused, respectively, and each row of psums is vertically accumulated. Since the number of 2D convolutions in a CONV layer is equal to the product of number of ifmap/filter channels, number of filters and fmap batch size, the logical PE array requires N×M×C logical PE sets to complete the processing of an entire CONV layer. Folding means mapping and then running multiple 1D convolution primitives from different logical PEs on the same physical PE. In the RS dataflow, folding is done at the granularity of logical PE sets for two reasons. First, it preserves intra-set convolutional reuse and psum accumulation at the array level as shown. Second, there exists more data reuse and psum accumulation opportunities across the N×M×C sets. Folding multiple logical PEs from the same position of different sets onto a single physical PE exploits input data reuse and psum accumulation at the RF level. Mapping multiple sets spatially across the physical PE array also exploits those opportunities at the array level. The amount of logical PE mapping depends on the size of RF and PE array and becomes an optimization problem to determine the best folding. After the first phase of folding as discussed above, the physical PE array can process a few logical PE sets, called a processing pass. However, a second phase of folding which is at the granularity of the processing passes is needed which is again determined by the global buffer size. The

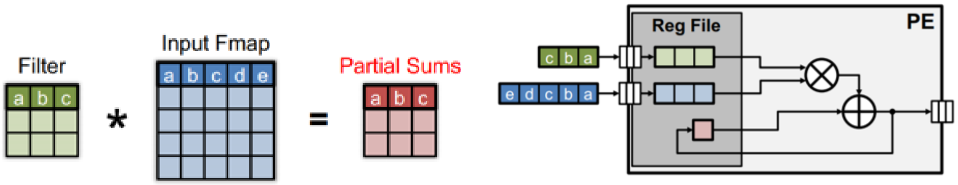


Figure 3.8. data movement of row stationary dataflow

figure 3.8 depicts the working of Row stationary dataflow. The initial *abc* from filter are placed in the register file. The activations from input F_{map} are sent in sequentially to the reg file. The computations are performed, and the partial sums are accumulated one at a time. The entire row of filter will be removed only after all the activations are parsed. The activations are also moved one pixel saving energy cost due to data movement. To maximize energy efficiency, the RS dataflow is built to optimize all types of data movement by maximizing the usage of the storage hierarchy, starting from the low-cost RF to the higher-cost array and global buffer. The way each level handles data is described as follows. By running multiple 1D convolution primitives in a PE after the first phase folding, the RF is used to exploit all types of data movements. Specifically, there are convolutional reuse within the computation of each primitive, filter reuse and ifmap reuse due to input data sharing between folded primitives, and psum accumulation within each primitive and across primitives. For inter-PE communication Convolutional reuse exists within each set and is completely exhausted up to this level. Filter reuse and ifmap reuse can be achieved by having multiple sets mapped spatially across the physical PE array. Psum accumulation is done within each set as well as across sets that are mapped spatially. Depending on its size, the global buffer is used to exploit the rest of filter reuse, ifmap reuse and psum accumulation that remain from the RF and array levels after the second phase folding. While the RS dataflow is designed for the processing of high-dimensional convolutions in the CONV layers, it can also support two other layer types naturally: The computation of FC layers is the same as CONV layers, but without convolutional data reuse. By swapping the MAC computation with a MAX comparison function in the ALU of each PE, the RS dataflow can also process POOL layers.

3.3.5 Evaluation

Different dataflows were compared using same number of PEs constraining the area. The baseline storage for a given number of PEs is given by

$$PE \times Area(512BRF) + Area(PE \times 512B) \text{ global buffer} \tag{3.1}$$

This baseline storage area is then used to calculate the size of the global buffer and RF in bytes for each dataflow. Following figure 3.9shows the data movement in both directions for all levels of hierarchy.

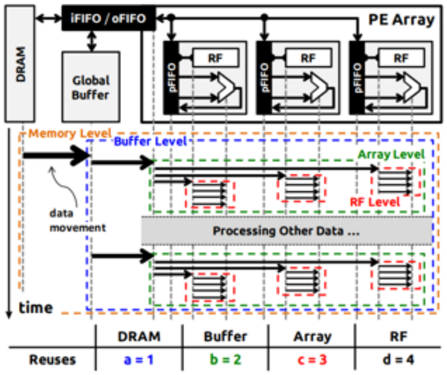


Figure 3.9. Energy cost for activation reuse

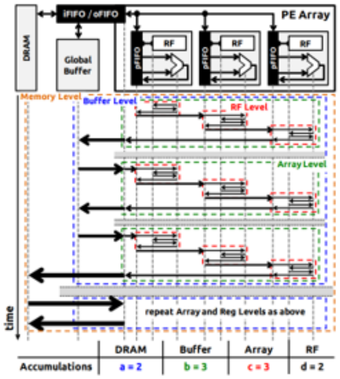


Figure 3.10. Energy cost for psum reuse

The energy cost for retrieving data from different levels is *DRAM*: 200, *global buffer*: 6, *inter array*: 2 and *rf*: 1. The energy of dataflow is formulated in two parts, input data access energy cost including filters, ifmaps and psum accumulation energy cost. If input value is used, it is brought from DRAM to RF once and reused many times. However, due to limited storage and operation scheduling the values are kicked out from the RF. So, we need to fetch from higher levels. Following this pattern as shown in figure 3.9 data reuse can be split across 4 levels. Suppose the total number of reuses for data is $a \times b \times c \times d$, the energy cost is

$$a * EC(DRAM) + ab * EC(\text{global buffer}) + abc * EC(\text{array}) + abcd * EC(rf) \tag{3.2}$$

Similarly as shown in figure 3.10, for psums, due to the overall operation scheduling, they must be stored to a higher-cost level and read back again afterwards. The energy cost is given by

$$2a - 1 * EC(DRAM) + 2a * b - 1 * EC(\text{global buffer}) + ab * c - 1 * EC(\text{array}) + 2abc * (d-1) * EC(rf) \tag{3.3}$$

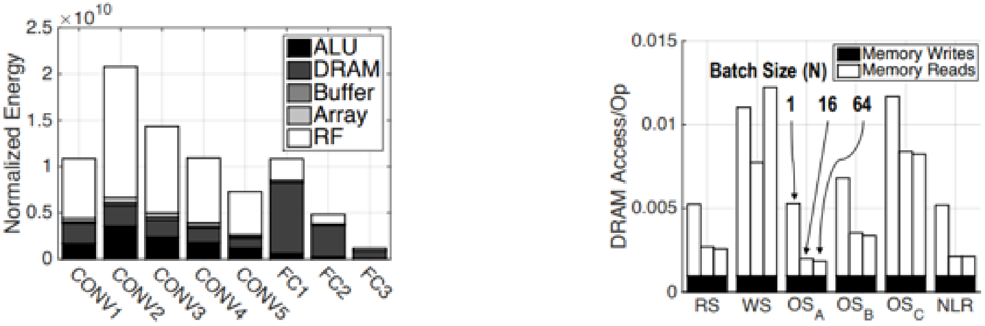


Figure 3.11. Normalized energy and DRAM access

Energy consumption of convolutional layers is dominated by RF access, showing that RS exploits different types of data movement in local RF minimizing access to storage levels of higher costs. While DRAM accesses dominate the energy consumption of FC layers due to the lack of convolutional data reuse. DRAM access has impact on overall energy usage. The writes are same as accelerator writes ofmaps. While RS, OSA, OSB and NLR have lower DRAM accesses than WS and OSC, as the former have more data reuse. RS has less on-chip storage compared to others showing that co design of architecture and dataflow is important. Overall, RS is 1.4 to 2.5 more energy efficient than other dataflows. Although OSA, OSB and NLR have similar or even lower DRAM accesses compared with RS, RS still consumes lower total energy by fully exploiting the lowest-cost data movement at the RF for all data types. NLR does not use the RF at all. Most of its data accesses come from the global buffer, which results in high energy as shown in figure 3.11. RS is the only dataflow that optimizes energy for all data types simultaneously.

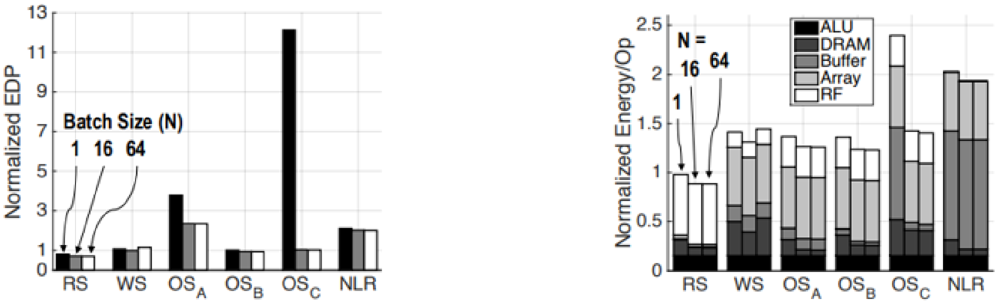


Figure 3.12. Energy delay product

Energy-delay product is used to verify that a dataflow does not achieve high energy efficiency by sacrificing processing parallelism. The delay is calculated as the reciprocal of number of active PEs. A dataflow may not utilize all available PEs due to the shape quantization effects and mapping constraints. RS has the lowest EDP since its mapping of 1D convolution primitives efficiently utilizes available PEs. The RS dataflow as shown in figure 3.12 is at least 1.3 more energy efficient than other dataflows at a batch size of 16 and can be up to 2.8 more energy efficient at a batch size of 256. Although throughput increases by more than 10 by increasing the number of PEs, the energy cost only increases by 13%. As a larger PE array also creates more data reuse opportunities. However, the trade-off between throughput and energy is not monotonic. The energy cost becomes higher when the PE array size is too small due to less reuse in the PE array and further increasing the global buffer does not contribute much to data reuse.

3.3.6 Summary

The energy costs of different CNN dataflow on spatial architectures are presented. Row Stationary is presented that minimizes energy consumption by maximizing input data reuse and minimizing partial sum accumulation cost simultaneously, and by accounting for the energy cost of different storage levels. Compared with existing dataflows such as the output stationary, weight stationary, and no local reuse dataflows, the RS dataflow is 1.4 to 2.5 more energy efficient in convolutional layers, and at least 1.3 more energy efficient in fully-connected layers for batch sizes of at least 16. Also, DRAM access alone wont dictate energy efficiency, dataflow to global buffer also impacts energy. For all dataflows, increasing the size of the PE array helps to improve the processing throughput at similar or better energy efficiency. Larger batch sizes also result in better energy efficiency in all dataflows except for WS, which suffers from insufficient global buffer size. Finally, for the RS dataflow, the area allocation between processing and storage has a limited effect on energy-efficiency, since more PEs allow for better data reuse, which balances out the effect of less on-chip storage.

3.4 Instruction Set Architecture

3.4.1 Introduction

In general accelerators adopt high level and informative instructions that directly specify the high-level functional block instead of low-level computational operations. However, they can be fully optimized for each instruction. Although its easy to implement for a small set of similar NN techniques, design complexity and the overhead of the instruction decoder for such accelerators will become large as the need of flexible support for a variety of NN results in a significant expansion of instruction set. Consequently, such designs support only a small subset of NN sharing very similar computational patterns and data locality but is incapable of handling the significant diversities[25]. As a result, the ISA design is still a fundamental yet unresolved challenge that greatly limits both flexibility and efficiency of existing NN accelerators.

In this section we try to understand how ISA can be designed for optimum performance. Initially it decomposes the instructions describing the high-level functional blocks into shorter instructions of low-level computational operations to allow the accelerator to have a broader application scope. And these low-level operations can be assembled to new high-level functional blocks. Also, the simple and short instructions can reduce the design complexity, power and area used by the instruction decoder.

Cambricon is a load-store architecture whose instructions are all 64-bit and contains 64 32-bit General-Purpose Registers (GPRs) for scalars, mainly for control and addressing purposes. To support intensive, contiguous, variable-length accesses to vector/matrix data with negligible overhead, it keeps data in on-chip scratchpad memory, which is visible to compilers. Implementation of multiple ports in the on-chip memory is not needed as simultaneous accesses to different banks decomposed with addresses' low-order bits are enough to supporting NN techniques. Unlike SIMD whose performance is restricted by the limited width of register file, Cambricon efficiently supports larger and variable data width because the banks of on-chip scratchpad memory can easily be made wider than the register file. There is negligible latency, power, and area overhead with a versatile coverage of 10 different NN benchmarks.

3.4.2 Overview of the proposed ISA

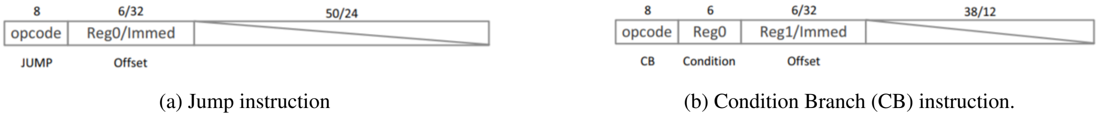
To design an efficient ISA, we need to analyze NN techniques in terms of computational power, memory access patterns. When accommodating these operations, data-level parallelism enabled by vector/matrix instructions can be more efficient than instruction-level parallelism of scalar instructions. a small yet representative set of vector/matrix instructions for existing NN techniques should be customized instead of simply re-implementing vector/matrix operations from an existing linear algebra library. Data access patterns for neural networks requires variable length and using fixed width is not efficient. So, the vector register files are replaced with on chip scratch pad memory, providing flexible width for each data access.

The Cambricon is a load store architecture which allows main memory to be accessed with

Table 3.3. An overview to Cambricon instructions

Instruction Type		Examples	Operands
Control		jump, conditional branch	register(scalar value), immediate
Data transfer	Matrix	matrix load/store/move	register(matrix address/size, scalar value), immediate
	Vector	vector load/store/move	register (vector address/size, scalar value), immediate
	Scalar	scalar load/store/move	register(scalar value), immediate
Computational	Matrix	matrix multiply vector, vector multiply matrix, matrix multiply scalar, outer product, matrix add matrix, matrix subtract matrix	register(matrix/vector address/size, scalar value)
	Vector	vector elementary arithmetics (add, subtract, multiply, divide), vector transcendental functions(exponential, logarithmic), dot product, random vector generator, maximum/minimum of a vector	register (vector address/size, scalar value)
	Scalar	scalar elementary arithmetics, scalar transcendental functions	register(scalar value), immediate
Logical	Vector	vector compare (grater than, equal), vector logical operations(and, or, inverter),vector greater than merge	register(vector address/size, scalar), immediate
	Scalar	scalar compare, scalar logical operarion	register(scalar), immediate

load/store instructions. It has 64 32-bit general purpose registers for using in register indirect addressing of the on-chip scratchpad memory. The Cambricon as shown in table 3.3 contains four types of instructions: computational, logical, control and data transfer instructions. The instruction length is fixed for memory alignment and design simplicity.



(a) Jump instruction

(b) Condition Branch (CB) instruction.

Figure 3.13. Jump instruction & Condition Branch (CB) instruction

Cambricon has two control instructions, jump and conditional branch as shown in figure 3.13. The jump instruction specified the offset via immediate or GPR value which is accumulated to the program counter. The conditional branch instruction specifies the predictor in addition to offset and branch target (PC + offset) is determined by a comparison between predictor and zero. Data transfer instructions support variable data size in order to flexibly support matrix and vector computational/logic instructions. These instructions support variable size data movement to/from main memory to on-chip scratchpads or scalar GPRs.

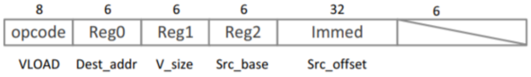


Figure 3.14. Vector Load (VLOAD) instruction.

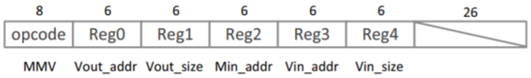


Figure 3.15. Matrix Mult Vector (MMV) instruction

Above instruction in figure 3.14 implements vector LOAD instruction, which loads a vector of size of V_size from main memory to scratchpad. And source address in main memory is sum of base address and offset saved in GPR. Cambricon keeps data in on-chip scratchpad memory, which is visible to compiler. The vector/matrix operations are variable sized and only restriction is that operands in the same instruction cannot exceed the capacity of scratchpad. If exceeded the compiler will divide them into shorter instructions. The memory capacity for vector instructions is 64KB and for matrix instruction is 768KB.

NNs can be naturally decomposed into scalar, vector, and matrix operations, and the ISA design must effectively take advantages of the potential data-level parallelism and data locality. Based on the existing NN techniques 6 matrix instructions are developed.

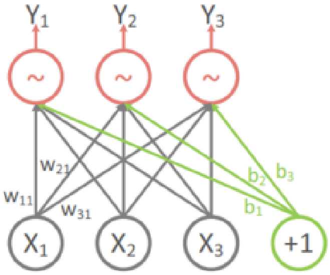
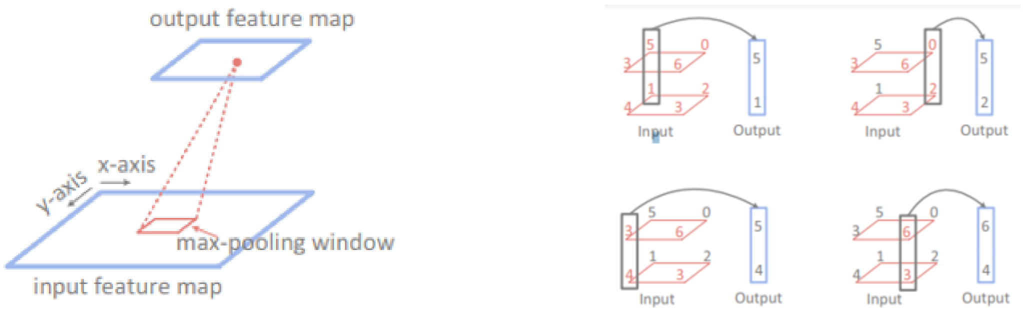


Figure 3.16. Typical operations in NNs.

The output of a neuron can be written as f(Wx+b). Where x and b are vectors of input neuron. W is the weight matrix and f are the element wise version of activation function f. The critical step of compute Wx is performed by Matrix-Mult-Vector(MMV) as shown in figure 3.16 where Reg0 base scratchpad memory address of the vector output. Reg1 specifies the size of the vector output, Reg2, Reg3, and Reg4 specify the base address of the matrix input, the base address of the vector input, and the size of the vector input. The Wx is computed using dedicated MMV instruction as shown in figure 3.15 instead of decomposing it as multiple vector dot products, as it needs overhead for synchronization, concurrent read/write requests to same address to reuse input vector x for different rows of M. MMV works for forward propagation and for backward propagation we can use same instruction along with additional instruction to implement the matrix transpose which is costly. A new instruction Vector-Mult-Matrix instruction is used directly for this purpose and has same fields with different opcode. While training weights are incrementally updated using $W = W + \eta(\delta W)$ where eta is the learning rate and delta W is the outer product of two vectors. Cambricon provides an Outer Product (OP) instruction, a Matrix-Mult-Scalar (MMS) instruction, and a Matrix-Add-Matrix (MAM) instruction to collaboratively perform the weight updating. Also, a Matrix-Subtract-Matrix (MSM) instruction is used to update weights in Restricted Boltzmann Machine.

While Vector-Add-Vector VAV instructions is used for vector addition, it requires multiple instructions to support the element-wise activation. In case of sigmoid function $f(a) = e^a/(1+e^a)$, the element wise activation is divided into three steps. Computation of the exponent $e^a i$ in vector a is done by Vector Exponential instruction VEXP. Constant 1 is added to all the elements using Vector-Add-Scalar VAS instruction. Dividing $e^a i$ by $1+e^a i$ is done by Vector-Dic-Vector VDV instruction. However, in order to implement other activation functions a series of vector arithmetic instructions, such as Vector-Mult-Vector (VMV), Vector-Sub-Vector (VSV), and Vector-Logarithm (VLOG), Random-Vector (RV) are provided. During the design of a hardware accelerator, instructions related to different transcendental functions can be implemented using same functional blocks.



(a)

(b)

Figure 3.17. Max-pooling operation.

Max pooling as shown in figure 3.17 is supported with a Vector-Greater-Than-Merge (VGTM) instruction shown in figure 3.18. VGTM assigns each output vector by comparing corresponding elements of the input vector-0 and vector-1. In addition to the vector computational instruction,

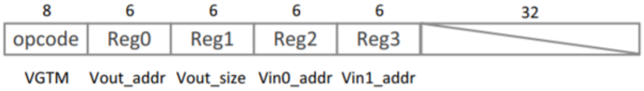


Figure 3.18. Vector Greater Than Merge (VGTM) instruction.

Cambricon also provides Vector-Greater-than (VGT), Vector-Equal instruction (VE), Vector AND/OR/NOT instructions (VAND/VOR/VNOT), scalar comparison, and scalar logical instructions to tackle branch conditions. 0.008% of arithmetic operations of NN cannot be supported by matrix and vector instructions and needs scalar operations as mentioned in the table. Following shows a code example for the implementation of NN components like the MLP feedforward layer, pooling layer, and Boltzmann machine layer using Cambricon instructions. The code density is significantly higher than X86 and MIPS.

3.4.3 A Prototype accelerator

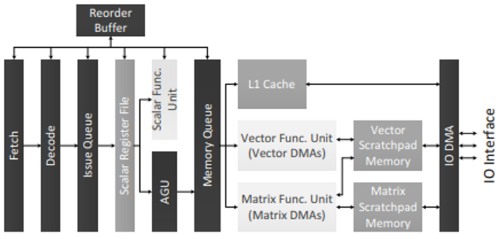


Figure 3.19. A prototype accelerator based on Cambricon.

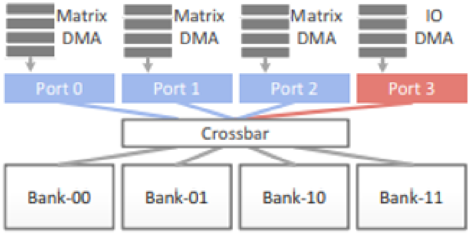


Figure 3.20. Structure of matrix scratchpad memory.

The accelerator in figure 3.19 has seven major instruction pipeline stages: fetching, decoding, issuing, register reading, execution, writing back, and committing. We use scratchpad and DMA as they fit the requirements of ISA. After the fetching and decoding stages, an instruction is injected into an in-order issue queue. After successfully fetching the operands from the scalar register file, an instruction will be sent to different units depending on the instruction type. Control instructions and scalar computational/logical instructions will be sent to the scalar functional unit for direct execution. After writing back to the scalar register file, such an instruction can be committed from the reorder buffer1 if it has become the oldest uncommitted yet executed instruction.

Data transfer instructions, vector/matrix computational instructions, and vector logical instructions, which may access the L1 cache or scratchpad memories, will be sent to the Address Generation Unit (AGU). These instructions must wait in an in-order memory queue to resolve potential memory dependencies with earlier instructions. After that, load/store requests of scalar data transfer instructions will be sent to the L1 cache, instructions for vectors, matrix will be sent to the vector, matrix functional units.

After execution, the instructions can be retired from memory queue and committed from the reorder buffer. The accelerator implements both vector and matrix functional units. The vector unit contains adders, multipliers, and scratchpad memory. The matrix unit contains multipliers and adders which are separated to avoid congestion and long-distance data movement. Cambricon does not use vector register files but keeps data in on-chip scratchpad memories and access data efficiently by using 3 DMAs. In addition, it has an IO DMA.

Scratchpad memory shown in figure 3.20 provides a single port for each bank but may need to address up to four concurrent read/write requests. So, the memory is decomposed into four banks according to addresses' low-order two bits, connect them with four read/write ports via a crossbar guaranteeing that no bank will be simultaneously accessed.

3.4.4 Evaluation

The ISA of DaDianNao contains four 512-bit VLIW instructions corresponding to fully connected, convolution, pooling, and local response normalization. And supported MLP, CNN, and RBN but fails to implement RNN, LSTM, AutoEncoder, Sparse AutoEncoder, BM, SOM and HNN as they are no the aggregates of the 4 supported layers. In contrast, Cambricon defines a total of 43 64-bit scalar/control/vector/matrix instructions and is sufficiently flexible to express all 10 networks. Code densities can be compared when the platform supports a wide range of applications.

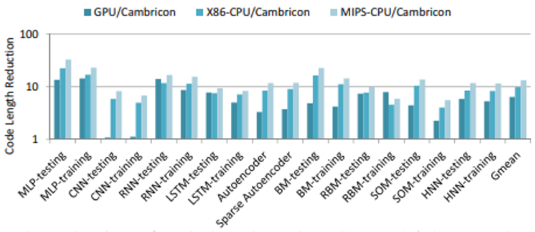


Figure 3.21. The reduction of code length against GPU, x86-CPU, and MIPS-CPU.

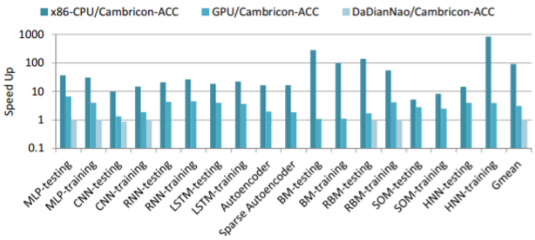


Figure 3.22. The speedup of Cambricon-ACC against x86-CPU, GPU, and DaDianNao.

On average as shown in figure 3.21, the code length of Cambricon is about 6.41x, 9.86x, and 13.38x shorter than GPU, x86, and MIPS, respectively. This is because the scalar operations are aggregated into vector instructions and vector operations into matrix instructions reducing code length. MLP has improved code density compared to CNN as aggregating scalar operations into vector operations has a small gain on code density. On average, 38.0% are data transfer, 4.8% are control, 12.6% are matrix, 33.8% are vector, and 10.9% are scalar instructions. So, vector/matrix instructions are critical for NN, and efficient implementations of these instruction improves performance of Cambricon based accelerators.

On average as shown in figure 3.22, Cambricon ACC is about 91.72x and 3.09x faster than of x86-CPU and GPU as it integrates dedicated functional units and scratchpad memory optimized for NN techniques. However, in comparison to DaDianNao, Cambricon is only 4.5% slower. This is because high level functions when broken down to lower level computation instructions brings in additional pipeline bubbles between instructions. But with high code density the amount of bubbles is minimal and incurs small loss. GPU consumes more energy as it spends excessive hardware resources to flexibly support various workloads. And Cambricon consumes slightly more than DaDianNao to accommodate the instruction pipeline logic, memory queue, as well as the vector transcendental functional unit. The overall area of Cambricon-ACC is 56.24 mm^2, which is about 1.6% larger than of DaDianNao.

3.4.5 Summary

Cambricon, a novel NN accelerator ISA to flexibly support a broad range of NN techniques was developed. Code density of Cambricon is significantly higher than that of x86 and MIPS and has area of 56.24 mm^2 and power consumption of 1.695W. Cambricon, this prototype accelerator can accommodate all ten benchmark NNs, while the state-of-the-art NN accelerator, DaDianNao, can only support 3 of them with negligible performance degradation.

3.5 Interconnects

3.5.1 Introduction

To solve the issue of energy dependency accelerators are mainly designed to be spatial in nature with interconnected processing elements to provide parallelism. The internal dataflow between the PEs is optimized to reuse parameters that are shared by multiple neurons. This reduces the number of memory accesses, [21] thereby providing energy efficiency. The PEs are fed new parameters from an on-chip global buffer as shown in figure 3.23.

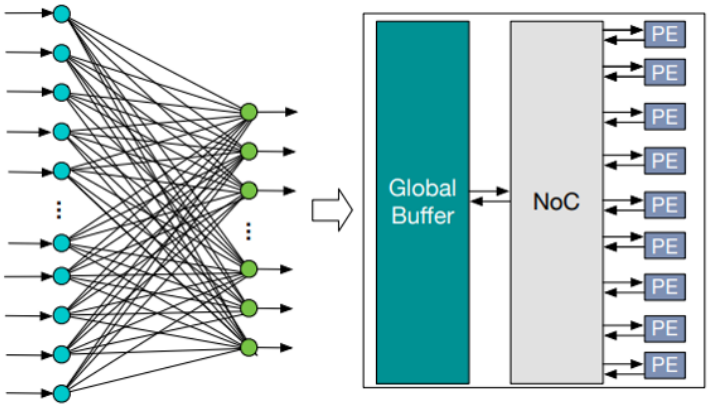


Figure 3.23. NoC is generated between the global buffer and the PEs using novel latency, area, and energy-efficient microswitches.

The microarchitecture of PE and nature of dataflow between PEs and global buffer – PEs is actively studied, however, implementation of the network-on-chip (NoC) interconnecting the PEs to each other and to the global buffer becomes a bottleneck as the compute of PE is improved. In a spatial NN accelerator, the NoC plays a key role, in realizing high throughput. This is because most spatial accelerators operate in a dataflow style: a PE operation is triggered by data arrival, and the PE stalls if the next data to be processed is unavailable due to memory or NoC delay. Almost all NN accelerators have used specialized buses, or mesh based NoCs, or crossbars, without a clear trade-off study on why one was picked over the other. A new NoC design paradigm for NN accelerators is designed using an array of reconfigurable micro-switches.

3.5.2 Overview

Most neural network accelerators employ processing elements which are primitive and manage computation for one partial sum of NN. Each PE contains some scratch pad memory and the compute logic. In addition to PEs there is also a larger on chip memory present in the accelerator, which we will refer to as the global buffer (GB). There are mainly three kinds of traffic flows in spatial accelerators: Scatter is data distribution from the GB to the PE array. Scatters can either be unicast or multicast, depending on the dataflow and the mapping of compute on PEs. Gather is the traffic flow which occurs when multiple PEs send data to the GB at a given interval of time. Gather can either occur at the end of the computation, or in the middle of the computation due to insufficient number of PEs. Local refers to the inter-PE communication traffic. It could be in the form of unicasts, multicasts or reductions. Figure 3.24 shows the total number of computations and communication flows within the layers of AlexNet. The raw compute to communication ratio across dataflows, demonstrates that communication is critical in spatial CNN accelerators to get full throughput. A delay in communication would essentially lead to a stall as PEs are tiny and are incapable of exploiting ILP/TLP mechanisms

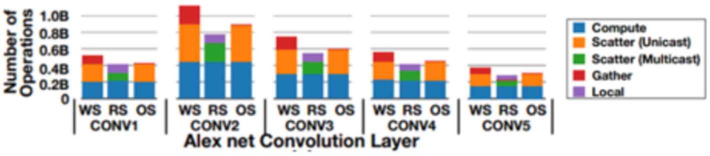


Figure 3.24. Compute vs.Communication of each Alexnet layer [14] across different CNN implementations: WS, RS, and OS.

for hiding delays. For all designs, scatter bandwidth is extremely crucial. For WS architectures, the bandwidth required by gathers is significant. As the number of PEs increases, so does the bandwidth across all traffic flows. Compute is dependent on the communication in accelerators and need interconnect to multiplex many neurons across finite PEs. The total traffic bandwidth divided by the PE delay determines the network bandwidth requirement per cycle to sustain full throughput, by Little's Law and this needs to be supported by NoC.

Traditional NoCs such as buses, meshes, and crossbars are common across multicores today. However, these NoCs add scalability challenges when used inside accelerators. Application specific NoCs generate NoCs in accordance with the application's communication graph. However, the traffic inside the accelerator is not static; it varies layer by layer and is dependent on the mapping of the dataflow over PEs, and the input parameters as shown in figure 3.25.

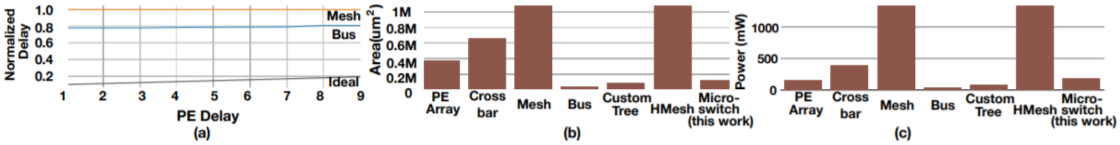
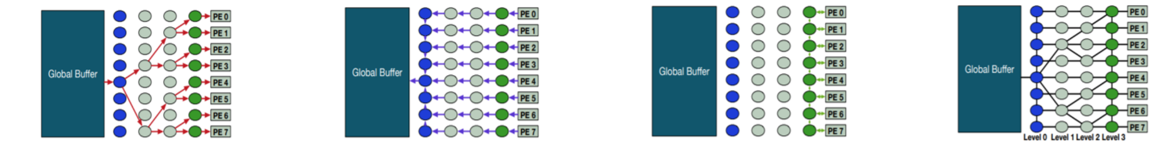


Figure 3.25. Challenges with traditional NoCs for accelerators. (a) Latency of 64-PE WS CNN accelerator with increasing PE delay (b) Area, and (c) Power

Performance of a mesh, and a multi-bus/multitree topology was compared against an "ideal" NoC which is a single-cycle zero contention network. It is observed that for a 1-cycle PE, the mesh and a single bus or tree is 10 slower than the ideal due to heavy contention at links near the GB. As PE delay increases the overall delay increases in ideal. However, for mesh or single bus/tree, the overall delay is almost constant showing that the NoC is the bottleneck. Traditional scalable networks like mesh routers, consume significantly higher area than even the compute PEs as routers in meshes are larger than a PE. Crossbars do not scale proportionately with number of nodes. Buses and the custom tree are better in terms of area. Power follows similar trend. So, meshes are not scalable solutions as NoCs inside accelerators as they get throughput limited when handling scatters and gathers and routers consume much higher area and power than PEs. Buses and trees are effective for an area and power, but they are non-configurable, limit the performance of accelerators across various domains.

3.5.3 Architecture

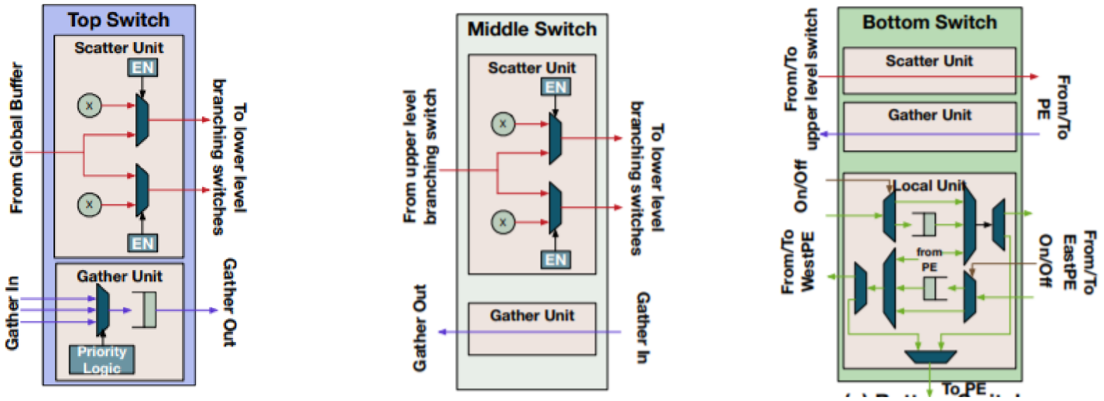
Accelerators achieve high throughput and energy efficiency in computation by distributing through processing engines, exploiting parallelism. Similarly, high network throughput is achieved by distributing communications to tiny microswitches. A microswitch consists of a small combinational circuit and up to two FIFOs; in contrast to the building blocks of traditional NoCs such as mesh routers that house buffers, a crossbar, arbiters, and control. The proposed NoC generator aggregates multiple microswitches and connects them in our proposed topology to build a light-weight interconnect, that can be plugged into NN accelerators. Multiple microswitches can be traversed within a single-cycle, enabling single-cycle communication inside the NoC.



(a) Scatter (Unicast/Multicast) (b) Gather (c) Local (d) Entire Connectivity

Figure 3.26. The connectivity of microswitch network for scatter, gather and local traffic.Top, middle, bottom switches shown by blue, gray and green colors

For a N PE design, we use a N log(N) microswitch array, as shown with log(N) levels, with N micro switches each. The design handles 3 dataflows as shown in figure 3.26: For scatters, we construct a tree structure in a microswitch array, with the root at one of the top switches, and the leaves at the bottom switches simulating the functionality of a bus delivering data to multiple destinations simultaneously within a cycle. It is implemented using two one-bit registers in each branching switch and control signal propagation wires. For gather, each PE has dedicated connections up to the top switches, via bypass links within the middle and bottom switches, providing high bandwidth. For PE to PE local traffic flows, a bi-directional linear network is designed using the bottom switches, as shown. This network allows single-cycle traversals between any two PEs by controlling the microswitches. The bandwidth of the scatter and gather networks is limited by the number of IO ports at the GB. The required bandwidth depends not just on the traffic, but also on the delay and context state in each PE, which comes as an input to the microswitch NoC generator.



(a) Top Switch

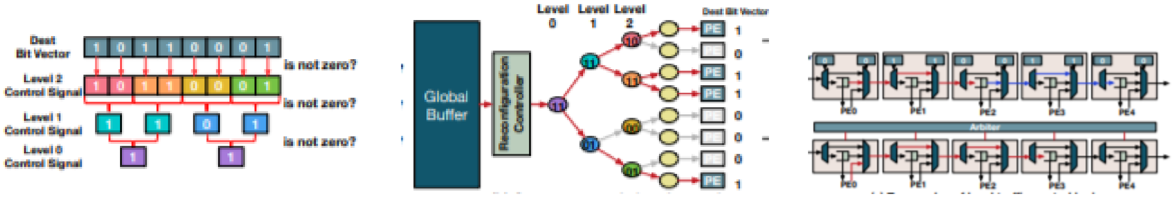
(b) Middle Switch

(c)) Bottom Switch

Figure 3.27. The microarchitecture of three microswitches.

The level of a microswitch is the number of layers between that microswitch and the global buffer. Each level has different kind of microswitch as there are different traffic patterns as shown in figure 3.27. Top switches manage the gather and scatter, from PE to global buffer and vice versa and has scatter and gather units. The scatter unit passes incoming flits to the branching nodes in the next level depending on the value of two one-bit control registers, determined by destinations of traversing flits. The traversal is completely buffer less, with flits branching based on unicast or multicast. The gather unit delivers incoming flits towards the global buffer I/O ports. A round-robin arbiter is used inside this unit to select one of the three gather flits. There is a FIFO after the arbiter to buffer the gather while it waits for arbitration at next micro switch. Middle switches manage scatter and gather traffic. The scatter unit is the same as that in top switches and the gather unit is just a wire that forwards incoming gather flits toward top switches. pipeline latches are inserted to meet the operating clock frequency, managed by generator. NoC can provide single-cycle traversals up to the top switches for most of NNs. Bottom switches, which belong to the last level, manage scatter, gather, and local traffic. While scatter and gather units are wires, the local unit consists of components like muxes, demuxes, FIFOs, and combinational logic that generates the control signals. Bottom switches increases linearly with the number of PEs. Single-cycle multi-hop designs require extra control logic to manage conflicts dynamically. Buffers are used at bottom switch if the destination PE is full, or the number of microswitches to be traversed may be greater than MPC_{max}.

The network interface between a PE and a bottom switch inserts a one-hot encoded bit vector that represents the number of remaining traversals. For scatters, the predetermined routing by the micro switch control logic enables single cycle communication. For gathers, the route of all flows is fixed. For local traffic, the NIC of source PEs inserts a one-hot bit vector representing the number of microswitches to traverse until the destination. Different traffic uses different flow-control strategies. For scatters, we employ a customized cycle-by-cycle circuit switching technique done by the network controller. The global buffer performs a scatter only if all destination PEs have at least one free buffer. For gather traffic, since the traffic passes through unidirectional wires, no flow control is required here. Top switches need a flow control for gathers, since an arbitration grant plus an empty FIFO slot in the next top switch is required before a flit can be dequeued. For local traffic, it supports Static, where the bottom switches are preset to enable multiple parallel circuit-switched connections between different PEs. In Dynamic, part of or the entire set of local links can be arbitrated for and used like a bus. Microswitch network has cycle-by-cycle reconfigurability which is controlled by one-bit control registers for muxes at each microswitch, to enable single-cycle traversals.



(a) Scatter control signal generation

(b) Scatter control signal mapping

(c) Two modes of local traffic control logic

Figure 3.28. A scatter tree reconfiguration. (a) Control signal generation. (b) Control signal mapping for a multicast scatter. (c) Local traffic control.

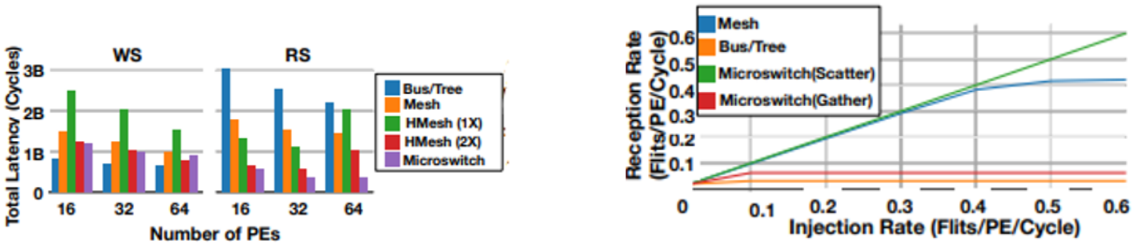
Gather network uses conventional flow-control and delivers flits in a pipelined manner. The reconfiguration for scatters is controlled by two one-bit control registers in each middle and top switch that are branching nodes in the tree. The network controller converts destination bits of a flit into control register values and sends them one cycle before the data flit traverses the scatter tree as shown in figure 3.28 which are pipelined to insert flit every cycle.

The controller receives a destination bit vector from the global buffer, that is a valid destination, and generates a control signal containing the value of control registers in branch switches of the scatter/ broadcast tree. The control signal logic is generated so that each branching switch can send a flit toward a lower branch which has at least one valid destination. The control signal is determined by examining two, four, and 2^k consecutive bits in a destination bit vector for the level log(N) – k. The logic checks if the destination bit vector is nonzero which results the control signals for the last level. In the next step, the consecutive two-bits are checked for nonzero values producing control signals for the next level. The logic repeats until the test bit size covers the half of the destination bit vector. If the number of PEs is not a power of two, zero padding is done for the invalid destinations. The local network partitions the set of local links into single-cycle circuit-switched paths between any two PEs. Since the network controller manages delivery of scatters, it also knows PE-PE communication, and accordingly tries to provide neighbor-to-neighbor communication, in parallel. Bottom microswitch has 2-bits to determine whether incoming flits need to be forwarded to the next microswitch. If stopped, the values are read by the appropriate PE if the destination matches making it configurable.

The number of bits in the control plane used to configure each microswitch has a trade-off with reconfiguration time, and multiple implementations can exist. In Dedicated, it uses 2.NlogN wires, to enable cycle by cycle reconfiguration. As an energy optimization, controller only sends bits to switches that need to update their configuration. In Ring an alternate design is used for configuration of ring to carry a switch id and the 2-bit configuration. The controller sends configurations for each switch in advance, assuming the delay of traversing the ring which is possible since the dataflow is fixed after the mapping is complete.

3.5.4 Evaluation

Mesh adds too much overhead in terms of area, and power, compared to the PE array. The crossbar area and power are reasonable at 32-64 PEs, but it shoots up at large PE counts. The mesh and crossbar consume 7.4X more power and 7.2X more area compared to the PE array at 256 PEs. The bus, tree and micro switch array are the most scalable for area and power. On average, the micro-switch array consumes 47.8% lower area and 39.2% lower power than all baselines.



(a) Total latency of each accelerator

(b) Throughput evaluation of different switches.

Figure 3.29. (a) Total latency of each accelerator and NoC combination for entire Alexnet. (b) Throughput evaluation of various microswitches.

Figure 3.29 shows the total cost of running AlexNet for WS and RS configurations. Multi scatter is dominant in WS traffic and bus, tree performs well with WS accelerators. However, as RS accelerators involve local traffic, the micro-switch network performs the best because it exploits the local traffic network between the bottom switches. Mesh performs the worst in every case because it needs to serialize all the scatter traffic. The performance of the microswitch scatter network scales linearly without saturating as it guarantees single-cycle traversal to multiple destinations. The microswitch gather network saturates early due to heavy congestion at the link going into the GB, however, wider links at the top switches to enhance throughput. The micro-switch fabric provides the lowest runtime, a 49% savings on average across all NoCs, as it eliminates the scatter or gather bandwidth bottlenecks present in other NoCs. Since a bus always broadcasts flits to the PE array, it requires more energy for each flit. Worst case would be for unicast where it has only one destination, but bus consumes energy for broadcast. As shown in figure 3.30, a greater number of PEs aggravate the energy inefficiency of bus. The second figure shows that micro-switch NoC being the most efficient in terms of overall energy as it activates only the required minimal links for each flit traversal, for both scatters and gathers. Depending on the operating clock frequency, the number of bottom micro-switches a local traffic flit can traverse within a cycle varies as shown in the figure. The MPC_{max} value affects the throughput of local traffic network based on the source destination pattern. If an accelerator design requires end-to-end local traffic, then the delay of such local traffic flits is the number of PEs divide by MPC_{max}.

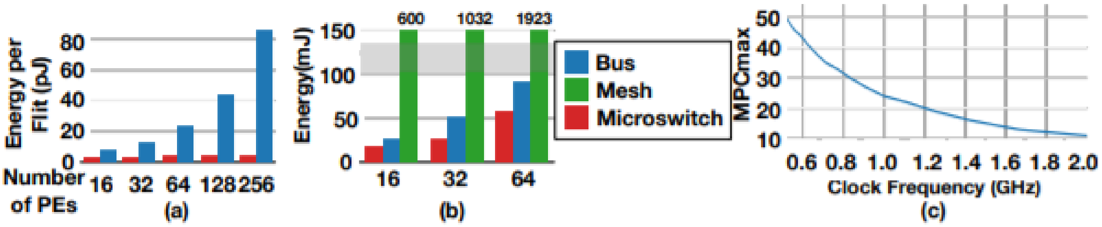


Figure 3.30. (a) Energy consumption for single flit traversal. (b) Total network energy using an RS accelerator (c) MPC$_{max} overclockfrequencyvalues$

3.5.5 Summary

A novel NoC design for neural network accelerators that consists of configurable light-weight micro-switches is designed. The microswitch network is a scalable solution for all the four aspects - latency, throughput, area, and energy - while traditional NoCs (bus/mesh/crossbar) only achieve scalability for some of them. We also provide a reconfiguration methodology to enable single-cycle paths over multiple micro-switches to support dynamism across neural network layers, mapping methodologies and input sizes. While our evaluations focused on neural network accelerators, we believe that the micro-switch fabric can be tuned for any accelerator built using a spatial array of hundreds of PEs.

3.6 Bit Level Optimization

3.6.1 Introduction

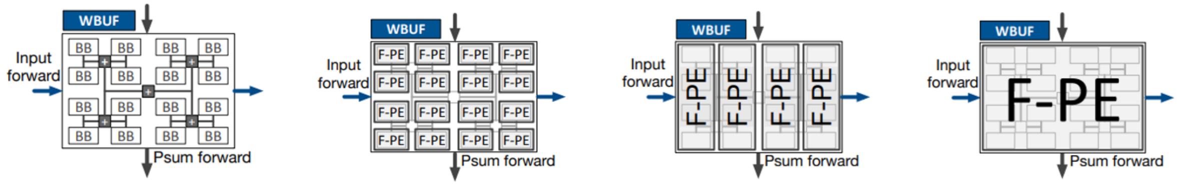
Data movement is a critical part of accelerator design and should be kept low as possible.One feature of DNNs is that the bitwidths can be reduced without the loss of accuracy. And the bit widths for each layer varies and using a fixed bit width would produce limited benefit or degradation. Bit Fusion architecture leverages this feature and introduces run time bit level fusion, decomposition to achieve better performance. This bit level flexibility helps us to minimize the computation at finest granularity possible with no loss of accuracy.

The number of bit level operations required for multiply and add is proportional to the product of the operands bit width and scales linearly with addition operator. So, matching the bit width of multiply and add units to reduce the bit width of DNN layer reduces the bit level computations almost quadratically. The energy consumption for DNN acceleration is dominated by on-chip and off-chip data access[32]. Bit fusion comes with encoding and memory access logic that stores and retrieves values in the lowest required bandwidth. Finally, DNNs can work with reduced bit width without the loss in accuracy.

3.6.2 Architecture

To minimize the computation, Bit fusion dynamically matches the architecture of the accelerator to the bitwidth required for DNN which may vary layer to layer without loss of accuracy. It is a collection of bit-level computational elements, called BitBricks, that dynamically compose to logically construct Fused Processing Engines (Fused-PE) that execute DNN operations with the required bitwidth.

Bit fusion arranges the bit bricks into physical groups called fusion units as shown in figure 3.31. Each bit brick can perform binary or ternary multiply or add operation. Bit bricks logically fuse together at runtime to form fused processing engines. The bit bricks in a fusion unit multiply a variable bitwidth input with weight to generate product and adds product to incoming partial



(a) Fusion Unit with 16 BitBricks (b) 16x Parallelism (c) 4x Parallelism (d) No Parallelism,8-bits

Figure 3.31. Dynamic composition of BitBricks (BBs) in a Fusion Unit to construct Fused Processing Engines (Fused-PE), shown as F-PE.

sum to produce an outgoing partial sum. In figure b each bit brick is logically divided into one fusion PE offering highest parallelism possible. In figure c we have four units combines to form one FPE and supports 8-bit input and 2-bit weights. The input bits can be varied by changing the spatial arrangement. Finally, when all bit bricks are fused together there is no parallelism. Dynamic composability of the fusion unit at bit level exposes max possible parallelism to match the DNN operands.

DNNs offer parallelism and benefit from increasing number of fusion units. So, we must minimize the overhead of control in the accelerator by maximizing the number of Fusion Units and minimizing the overhead of dynamically constructing Fused-PEs. Energy consumption is dominated by data access and it needs to be reduced.

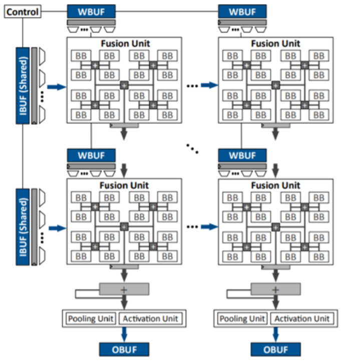


Figure 3.32. Bit Fusion systolic architecture comprising a collection of BitBricks (BBs) that can fuse to form Fused-PEs.

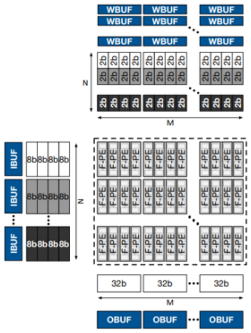


Figure 3.33. Bit-Flexible matrix-vector multiplication.

The systolic organization reduces the overhead of control by sharing the control logic across the entire systolic array as shown in figure 3.32. Systolic architecture fits the greatest number of bit bricks in the given area budget. Systolic organization improves sharing of input across columns and accumulates partial results across rows to minimize access to on chip memory. IBUF feed into rows parallelly and OBUF collects output , which is accumulated by each column accumulator, eliminating need for local buffers. Each fusion unit is accompanied by one weight buffer reducing on chip data access.

Depending on the number of Fused-PEs and their organization, the buffers must supply different number of operands with various bitwidth. Data is augmented in registers which are then infused using a series of multiplexers. The benefits of this design are reduced access to data array. As seen in the figure 3.33 an input vector of 4 × N 8-bit elements are multiplied by 4 × N × M 2-bit elements. To accommodate this the 16-bit bricks in fusion unit are logically composed to form four 8 × 2 fused PEs. Both input and weight buffer provide 32 bits per access. In this case input values are shared, and weights are specific to fused PEs. By reducing the bit widths of weight buffer, the parallelism is improved. Finally, the output is added to the incoming partial sum to produce the output partial sum.

Microarchitecture

The key insight that enables bit level composability is a multiply operation between operands with power of 2 bits can be decomposed into 2-bit multiplications which could be put through shift and add to produce the actual result. So, based on the operand size the we have the number of multipliers decomposed. This is how the smallest bitwidth of 2-bits is obtained. A single

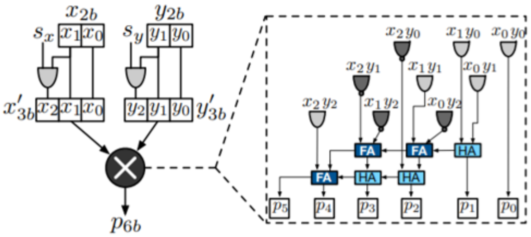


Figure 3.34. A single BitBrick. (HA: Half Adder, FA: Full Adder.)

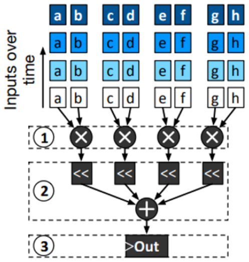
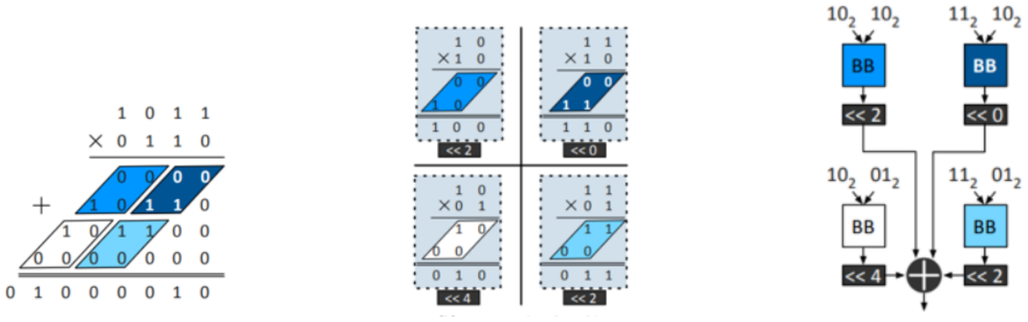


Figure 3.35. Spatial fusion. Operands ah are 2-bit

bit brick takes two 2-bit operands and two corresponding signs bits as shown in figure 3.34. According to sign bits the bit bricks extend the 2-bits to 3-bit signed multiplier to generate 6-bit product. Thus, bit brick support both signed and unsigned numbers as input. The bit width for the operand dictate how the results from the decomposed multiplication are left shifted before adding. In the below figure 3.36 we have two 4-bit multiplications which are decomposed into four 2-bit multiplication. After multiplication, these values are shifted based on operand bitwidth to find the result.



(a) A 4-bit multiplication($6_{10}\times 11_{10} = 66_{10}$)

(b) Decomposing the 4-bit multiplication to four 2-bit multiplications.

(c) Mapping decomposed multiplication to BitBricks(BBs).

Figure 3.36. Using BitBricks to execute 4-bit multiplications

Bit fusion can support up to 16-bit operands by first recursively breaking down the 16-bit multiplication to 8-bit, 4-bit, and 2-bit multiplication which execute using bit bricks. For multiplication between 2n-bit operands we have figure , if it is for operands of different sizes, then we have the second one.

$$\begin{aligned} A_{2n} &= 2^n \times (A_{2n})_{hi} + 2^0 \times (A_{2n})_{lo} \\ B_{2n} &= 2^n \times (B_{2n})_{hi} + 2^0 \times (B_{2n})_{lo} \\ A_{2n} \times B_{2n} &= 2^{2n} \times (A_{2n})_{hi} \times (B_{2n})_{hi} + 2^n \times (A_{2n})_{hi} \times (B_{2n})_{lo} + 2^n \times \\ &(A_{2n})_{lo} \times (B_{2n})_{hi} 4 + 2^0 \times (A_{2n})_{lo} \times (B_{2n})_{lo} \\ A_{2n} \times B_n &= 2^n \times (A_{2n})_{hi} \times B_n + 2^0 \times (A_{2n})_{lo} \times B_n \end{aligned} \tag{3.4}$$

Each level of recursion, from 16-bits to 8-bits, 8-bits to 4-bits, and 4-bits to 2-bits, requires additional shift-add logic. The overhead from the shift-add logic represents the hardware cost of bit-level flexibility. In fusion microarchitecture, the decomposed products generated by multiple bricks over a cycle are combined spatially. In temporal design, multiplication follows three steps 2-bit multiplication, shifting and accumulation. Even though this takes less hardware with the shifter and accumulator size depends on the higher bit width supported and limits the benefit.

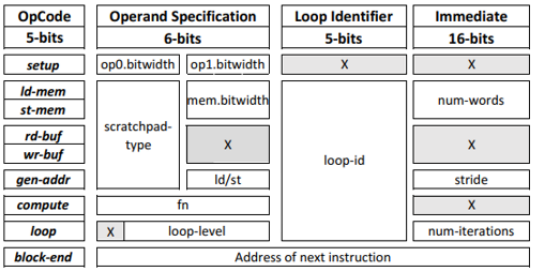
Spatial fusion in figure 3.35 follows the same steps; however, it improves upon the temporal design by using a shift-add tree and a single shared accumulator to reduce the number of gates required. Spatial multiplier uses more area than temporal but delivers more performance and the performance per area is more. Using spatial would improves bit bricks per area but the data access to SRAM buffers increases the area. So, a tradeoff is made to use both.

To leverage bit level flexibility a new hardware software interface is needed. To avoid the overhead of fine-grained control over operations at such a scale, the abstraction needs to amortize the cost of bit level fusion across blocks of instruction that implement the layer. Table 3.4 shows the different instructions available. The address is calculated in the following way

$$address = base + \sum_{id}(loop\_iterator[id] \times stride[id]) \tag{3.5}$$

The ISA is made flexible to enable layer specific optimizations.

Table 3.4. Bit Fusion Instruction Set



OpCode	**Operand Specification**		**Loop Identifier**	**Immediate**
5-bits	**6-bits**		**5-bits**	**16-bits**
setup	op0.bitwidth	op1.bitwidth	X	X
ld-mem ***st-mem***	scratchpad-type	mem.bitwidth	loop-id	num-words
rd-buf ***wr-buf***		X		X
gen-addr		ld/st		stride
compute	fn			X
loop	X	loop-level		num-iterations
block-end	Address of next instruction			

3.6.3 Evaluation

On average, Bit Fusion delivers 3.9 speedup since the Bit Fusion architecture can perform more DNN operations with lower bitwidth in each area compared to Eyeriss. The CNN benchmarks see higher performance gains compared to recurrent networks as convolution operations are more amenable for data reuse in systolic architecture of bit fusion. Architectures which can be computed using smallest bit width and operations that provide a large degree of parallelism exploit the increased number of fused PEs.

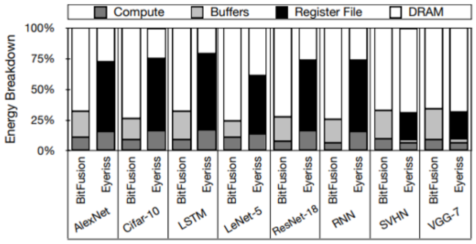


Figure 3.37. Energy breakdown of Bit Fusion and Eyeriss

Both architectures consume more than 80% of energy for on chip and off chip memory access. The bit level flexibility allows buffers to hold more data on chip reducing the number of off chip access. Since it enforces explicit data sharing for inputs and partial results and saves on register file access but requires more SRAM access. Both bit level flexibility and systolic organization of bit bricks in the bit fusion architecture reduced energy by 5.1X.

Depending on DNN topology as shown in figure 3.37, the impact of off chip bandwidth on performance varies. When bandwidth is 4X bit fusion provides 1.6X speedup and 60% degradation for 0.25X bandwidth. Batching amortizes the cost of weight reads by sharing weights across a batch of inputs. While GPUs can benefit from using as low as 8-bits, Bit Fusion can extract performance benefits for as low as 2-bit operations. It uses 16X speed up over TX2. On average bit fusion provides 2.6X speedup over stripes as stripes uses bit-serial computations to support variable bit widths just for DNN weights. Owing to this bit reduction energy is also reduced by 3.9X.

3.6.4 Summary

Deep neural networks use abundant computation but can withstand very low bitwidth operations without any loss in accuracy. Leveraging this property of DNNs, we develop Bit Fusion, a bit-level dynamically composable architecture, for their efficient acceleration. ISA was introduced to enable the software to utilize this bit level fusion capability to maximize the parallelism in computations and minimize the data transfer in the finest granularity possible. Bit Fusion achieves significant speedup and energy benefits compared to state-of-the-art accelerators.

3.7 Compressed Sparse Networks

3.7.1 Introduction

In general, common networks have significant redundancy and can be pruned dramatically during training without substantively affecting accuracy. There could be many weights that can be eliminated which varies from 20% to 80%. Eliminating weights results in a network with a substantial number of zero values, which can potentially reduce the computational requirements of inference[29]. There are further optimization opportunities. The nonlinear function converts negative numbers to zero and this output comes by multiplying activations with weights. This contribute to around 50-70% of activations and reducing this computation would improve performance. Additional benefits can be achieved by a compressed encoding for zero weights and activations, thus allowing more to fit in on-chip RAM and eliminating energy-costly DRAM accesses.

In this section we see how to exploit both weight and activation sparsity to improve the performance and power of DNNs. They only transfer data that can be multiplied and to reduce data access input stationary fashion reuse if done. To improve performance PE are run in parallel. The compression and tiling of the CNN data enables two energy-saving optimizations. Weight, activations are maintained in compressed form reducing energy-hungry data staging and transmission cost.

3.7.2 Dataflow

While the inner core of the dataflow in SCNN is based on spatial cartesian product, the complete data flow needs deep nested loop structure, mapped both spatially and temporally across multiple PE called planar tiled Input stationary cartesian product sparse. A CNNs dataflow defines how loops are ordered, partitioned and parallelized. PT-IS-CP sparse dataflow is selected because it enables reuse patterns that exploit the characteristics of sparse weights and activations.

There are different components to the dataflow, the IS term in PT-IS-CP describes the temporal part of the dataflow. The single multiplier temporal dataflow employs an input stationary computation order. This maximizes reuse of input while paying cost to stream the weights to the computation units. PT-IS-CP dataflow requires input buffers for weights and activations and accumulator buffer to store partial sums. In this dataflow we must perform read-add-write operation for every access, so we need an accumulator buffer along with attached adder called as an accumulation unit. The stationarity of input activations comes at the cost of more streaming accesses to the weights and to the partial sums in the accumulator buffer. Blocking the weights in the output channel improves efficiency, therefore we divide the output channels into groups inside the weight and accumulation buffers. Each iteration of this outer loop will require the weight buffer to be refilled and the accumulator buffer to be drained and cleared, while the contents of the input buffer will be fully reused because the same input activations are used across all output channels.

The CP term describes how parallelism of many multipliers within a PE can be exploited while maximizing spatial reuse. The intra PE parallelism fetches a vector of F filter and I inputs from the buffers and delivered to multipliers to compute cartesian product of partial sums. This allows for proper reuse of inputs. Each multiplier output is accumulated with a partial sum at the matching output coordinates in the output activation space. The final adder should have similar width as multiplier to match throughput.

Finally, the PT term in PT-IS-CP-dense describes how to scale beyond the practical limits of multiplier count and buffer sizes within a PE. Spatial tiling strategy is used to spread the work across all the PE. Unfortunately, strict partitioning both inputs and outputs does not work because of the sliding window nature of convolution operation introduces cross tile dependency at tile edge. These data halos are resolved in the following ways. In input halos the input buffers are sized slightly larger to accommodate halos. These halo input values are replicated across adjacent PEs, but outputs are strictly private to PE. The accumulation buffers at each PE are sized to be slightly larger and the halos now contain incomplete partial sums that must be communicated to neighbor PEs for accumulation.

The PT-IS-CP dataflow discussed till now is dense dataflow, the sparse version of this exploit's sparsity in weights and activations. The key feature is that decoding a sparse format ultimately yields a non-zero data value and an index indicating the coordinates of the value in the weight or input activation matrices. At each access the buffer delivers a vector of input and weights along with each of their coordinates within the region. The multiplier array computes in similar fashion, however, unlike dense architecture output coordinates are not derived from loop indices in a state machine but from the coordinates of non-zero values embedded in the compressed format. Also, the monolithic accumulation buffer is converted into a distributed array of smaller accumulation buffers and can be implemented using crossbar. The scatter network routes array of partial sums to an array of accumulator banks based on output index. Finally, when computation is done the accumulator buffer is drained and compressed into output buffer.

3.7.3 Architecture

As the convolution layers typically dominate both arithmetic and computation time, the SCNN architecture is optimized for efficiency on these layers.

A full architecture consists of multiple PEs connected via simple interconnect as shown in figure 3.38. The PE are connected to neighbors to exchange halo values during processing. The PE array is driven by a layer sequencer that controls the movement of inputs and is connected to DRAM controller to broadcasts weights through a arbitrated bus.

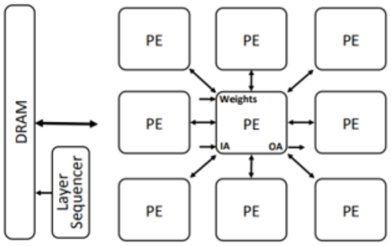


Figure 3.38. Complete SCNN architecture

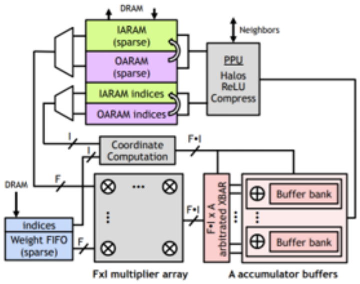


Figure 3.39. SCNN PE employing the PT-IS-CP-sparse dataflow.

To process the first CNN layer, the layer sequencer streams a portion of the input image into the IARAM of each PE and broadcast the compressed sparse weights into weight buffers. Once all layers are completed sparse-compressed output activation is distributed across the OARAMs of the PEs. The OARAM and IARAM are logically swapped between layers. Each PE state machine operate on weights and activations in order defined by the dataflow as shown in figure 3.39. First vector F of compressed weights and I of activations are fetched from the buffer and distributed across the multiplier array to compute the output partial sums. At the same time indices are processed to compute the output coordinates in the dense output activation space.

The FXI products are delivered to an array of accumulator banks indexed by output array. To reduce contention among products the bank size is larger than FXI. Each accumulator bank includes adders and buffers are double buffered to simultaneously update input and output buffers. When output channel group is complete post processing unit does the following. First, exchange partial sums with neighbor PEs for halo regions at boundary of PE output. Apply the nonlinear activations, pooling, and dropout functions and finally compress the output activations into the compressed-sparse form and write them into the OARAM. image

The encoding includes a data vector consisting of non-zero values and an index vector that includes the number of non-zero values followed by number of zeros before each value as shown in figure 3.40. Unlike convolutional layers, fully connected layers are not reused across multiple input activations. The CP approach does not align nonzero weights and activations that

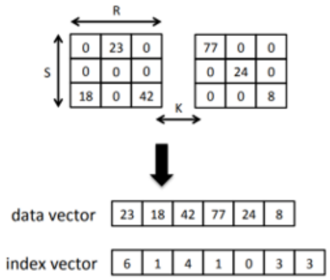
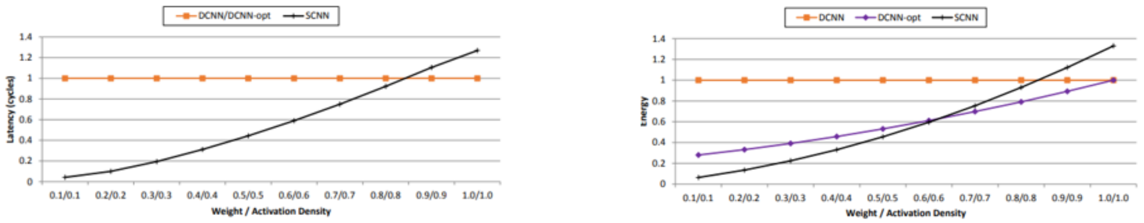


Figure 3.40. Weight compression.

must be multiplied, so the NXN SCNN multiplier operates at N multiplies per cycle (25% peak throughput). Similar for weights, so some additional multiplexing hardware is required to move nonzero weights into position. For large models' activations to be saved to and restored from DRAM. SCNN can temporally tile the activation space so that the collection of PEs operates on a sub-volume of the activations at a time. This temporal tiling can be applied in addition to the spatial tiling that SCNN already employs to partition the activation volume across the PEs.

3.7.4 Evaluation

Weight and activation are swept from 0 to 100% to observe the sensitivity to CNN sparsity. At full density, SCNN only achieve about 79% of the performance of DCNN because SCNN dataflow is susceptible to multiplier under utilization effects than DCNNs dot product flow. As density starts to reduce SCNN outperforms DCNN achieving 24X perf at 10%. At full density SCNN consumes 33% more energy due to the overheads of storing and maintaining the sparse data structure as shown in figure 3.41. However, as we reduce, we start to see the SCNNs outperforming.



(a) Performance

(b) Energy

Figure 3.41. GoogLeNet performance and energy versus density.

There is a performance gap between the actual and upper bound of SCNN. They suffer from intra-PE fragmentation when layers do not have enough useful work to fully populate the vectorized arithmetic units. Second, The PEs effectively perform an inter-PE synchronization barrier at the boundaries of output-channel groups which can cause early-finishing PEs to idle while waiting for laggards. The multipliers are under utilized especially in the last two layers as input activation volume reduces with time.

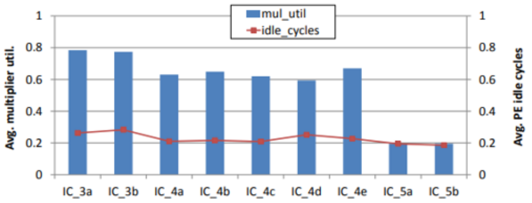


Figure 3.42. Average multiplier array utilization and fraction of time PEs are stalled on a global barrier.

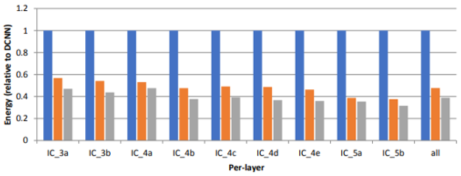


Figure 3.43. SCNN energy-efficiency comparison.

SCNN improves efficiency on an average by 2.3%, however it varies with the densities of layers as shown in figure 3.42 and 3.43. But for few networks where there is 100% activation density it poses a challenge where the overheads like crossbar and distributed accumulation RAMs overshadow any benefit from fewer arithmetic operations. As discussed, both cross PE global barriers and intra PE multiplier array fragmentation can degrade performance with former being more critical. SCNN-Sparse A is slightly more energy-efficient because of the removal of overheads (weight FIFO) to manage sparse weights. The input-stationary temporal loop around the Cartesian product makes these architectures extremely effective at filtering IARAM accesses, resulting in the IARAM consuming less than 1% of the total energy.

3.7.5 Summary

SCNN is used for inference in CNNs. It exploits sparsity in both weights and activations using the sparse planar-tiled input-stationary Cartesian product (PT-IS-CP-sparse) dataflow which helps to reuse weights and activations. Also, it allows for compressed representation for both weights and activations which allows for reduced data movement and increased on die storage capacity. For similar area SCNN beats energy optimized dense architecture when weights and activation density is less than 85%. For SOTA networks SCNN achieves performance improvement by factor of 2.7X and energy efficiency of 2.3X.

3.8 In Memory Computation

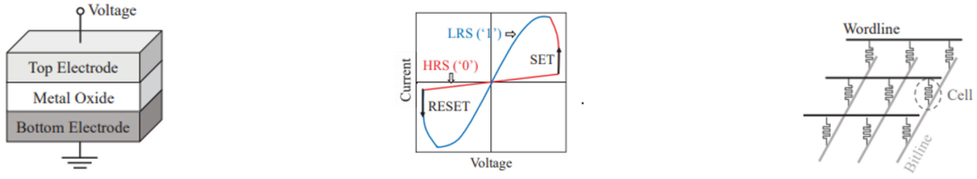
3.8.1 Introduction

Conventional computer systems adapt separate processing unit and data storage component. As volume of data to process has increased, the data transfer to and for memory has become critical performance bottle neck. Recent nonvolatile memory like ReRAM, STT-RAM, PCM have ability to perform logic beyond data storage and this processing in memory offers a promising solution for the memory wall issue. Among them ReRAM performs matrix-vector multiplication in a crossbar structure, and this [6] can represent synapses in neural computation.

Many adopted large on chip memory to store synaptic weights. Although this reduces the data transfer of synaptic weights, the movement of inputs and outputs is still a hindrance. So, a new architecture PRIME "Processing in ReRAM based main memory" which uses the efficiency of ReRAM and PIM techniques is proposed to perform computations in memory. A portion of the memory array is reconfigured to serve as neural accelerator besides normal memory.

3.8.2 Overview

Resistive random-access memory stores values by changing cell resistances.



(a) Conceptual view of a ReRAM cell; (b) I-V curve of bipolar switching; (c) schematic view of a crossbar architecture.

Figure 3.44

As shown in figure 3.44 by applying an external voltage on it the ReRAM cell can be switched between High and low Resistance states which represent logic 1 and 0. From the IV graph, by applying voltage we SET the value from 0 to 1 and RESET value from 1 to 0. ReRAM in cross bar structure have high density and have read and write latency like DRAMs. Image 3.45 shows a 2x2 NN implemented using crossbar. The input data ai is represented by analog input voltages on the word lines. The synaptic weights wij is programmed into the cell conductance in the crossbar array. Then the current flowing to the end of each bit line is viewed as the result of the matrix-vector multiplication

$$b_j = \sigma(\sum_{\forall i} a_i.w_{i,j}) \tag{3.6}$$

After sensing the current on each bit line, the neural networks adopt a nonlinear function unit to complete the execution. ADC and DAC are needed for analog computing. PRIME is a morphable ReRAM where a portion of the cross bars are enabled with NN computation function called full function subarray. While applications are running the crossbar is used for computation and when not in use the subarrays can be freed to provide extra memory capacity. Data transfer, DRAM access in general consume 95% of the total energy in state of art architectures. By implementing processing in memory, we take advantage of large internal bandwidth of main memory and make the data movement minimal. Instead of adding logic to memory, PRIME utilizes the memory arrays for computation making the area over head minimal.

3.8.3 Architecture

While most NN acceleration approaches require additional processing units, PRIME directly uses ReRAM cells to perform computation without using PUs. To achieve this, PRIME partitions a ReRAM bank into three regions memory sub array, full function sub array and buffer sub arrays as shown in figure 3.46.

The mem sub arrays only have data storage capability. The FF subarrays have both computation and data storage capabilities, and they can operate in two modes. The FF subarrays have

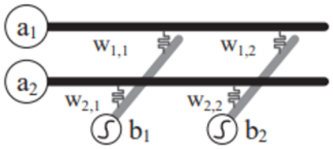


Figure 3.45. using a ReRAM crossbar array for neural computation.

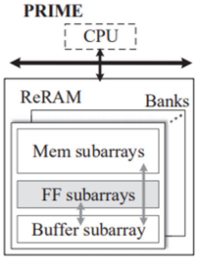
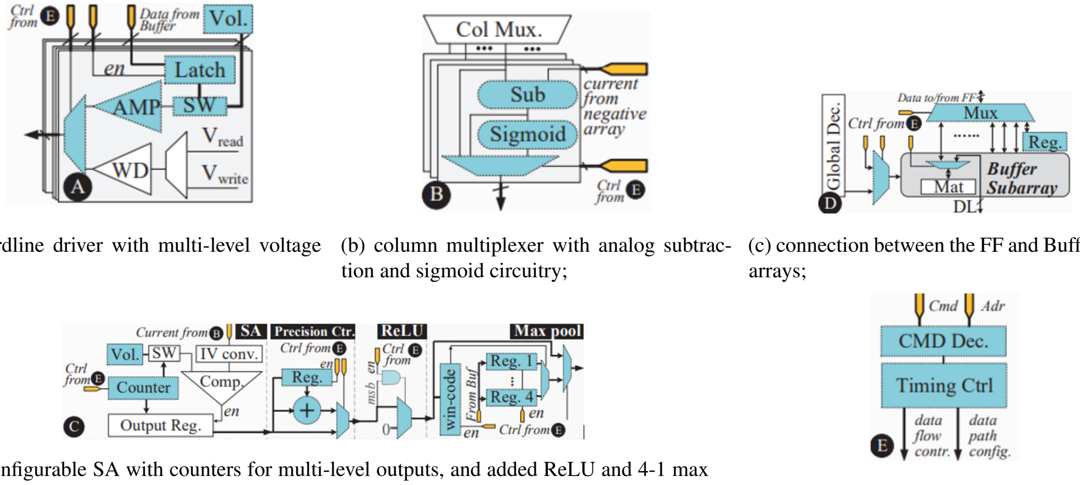


Figure 3.46. PRIME design.

both computation and data storage capabilities, and they can operate in two modes. In memory mode, it works as conventional memory and in computation mode they execute NN computations. The reconfiguration of FF subarrays is controlled by a PRIME controller. The buffer sub arrays serve as data buffer for the FF subarrays. They are connected to the FF subarrays through private data ports, so that buffer accesses do not consume the bandwidth of the Mem subarrays. For NN computation the FF subarrays enjoy the high bandwidth of in-memory data movement, and can work in parallel with CPU, with the help of the Buffer subarrays.

The design goal of FF subarray is to support both storage and computation with minimum area overhead. First, we start by adding several components in decoders and drivers. we attach multi-level voltage sources to the word lines to provide accurate input voltages determined by control signals. Latch is used to control simultaneous feeding of input data. Second, to drive the analog signals transferring on the word lines, we employ a separate current amplifier on each word line. We need 2^P in levels of input voltage. A mux is used to switch between memory and compute mode. Also, we employ two crossbar arrays store positive and negative weights, respectively, and allow them to share the same input port.

We need to modify the column multiplexers as shown in the figure 3.47. An analog subtraction unit and a nonlinear threshold (sigmoid) unit are incorporated. In few cases like when a large NN is mapped to multiple crossbar arrays, we bypass the sigmoid unit. In order to allow FF subarrays to switch bit lines between memory and computation modes, we attach a multiplexer to each bit line to control the switch. We only need to modify half of the column multiplexers



(a) Wordline driver with multi-level voltage sources

(b) column multiplexer with analog subtraction and sigmoid circuitry;

(c) connection between the FF and Buffer subarrays;

(d) reconfigurable SA with counters for multi-level outputs, and added ReLU and 4-1 max pooling function units;

(e) PRIME controller

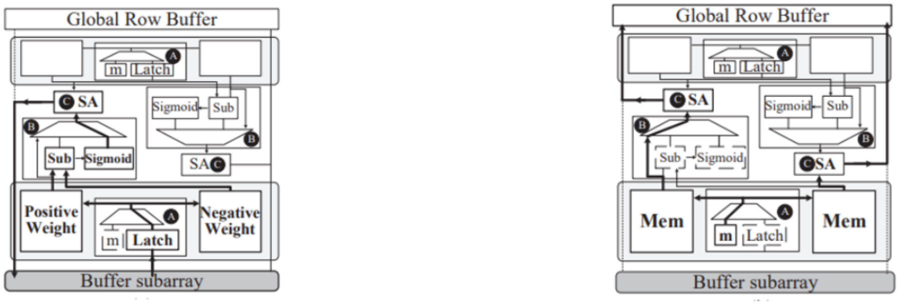
Figure 3.47. The PRIME architecture: functional blocks modified/added in PRIME

and after analog processing, the output current is sensed by local SA's. In above figure 3.47, NN computation requires Po bit $(Po \leq 8)$ precision reconfigurable SAs to offer much higher precision than memory does. Next, we configure it with value between 1 and po bits, controlled by counter and result is stored in output register. we then allow low precision ReRAM cells to perform NN computation with a high-precision weight, using a precision control circuit that with a register and an adder. Hardware unit for ReLU is supported, the circuit checks the sign bit and outputs zero if negative and result itself otherwise.

We enable an FF subarray to access any physical location in a Buffer subarray to accommodate the random memory access pattern in NN computation. Extra decoders and multiplexers are employed in buffer connection unit. In cases when output of one mat is input of other, we bypass the buffer and use registers for intermediate data storage. We reuse SAs and write drivers to serve ADC and DAC functions by slightly modifying the circuit design.

We must morph between two modes. In below figure 3.48 we have both computation mode(a) and memory mode (b). The bold line depicts data flow in both the cases. In computation mode, the FF subarray fetches the input data of the NN from the Buffer subarray into the latch of the word line decoder and driver. After the computation in the crossbar arrays that store positive and negative weights, their output signals are fed into the subtraction unit, and then the difference signal goes into the sigmoid unit. The analog output is converted to digital signal by the SA is written back to the Buffer subarray.

In memory mode, the input comes from the read/write voltage selection, and the output bypasses the subtraction and sigmoid units. Several steps are involved in morphing between memory and compute phases. PRIME migrates the data stored in the FF subarrays to certain allocated space in Mem subarrays, and then writes the synaptic weights to be used by computation into the FF subarrays. Then peripheral circuits are reconfigured by the PRIME controller, and the FF subarrays are switched to computation mode and can start to execute the mapped NNs. After computation tasks, the FF subarrays are switched back to memory mode through a wrap-up step that reconfigures the peripheral circuits.



(a) Computation mode;

(b) memory mode

Figure 3.48. An example of the configurations of FF subarrays.

The Buffer sub arrays are used to cache the input and output data from FF subarrays. The FF subarray can communicate with the buffer sub arrays directly without the involvement of CPU, improving parallelism. These are configured close to memory sub array to reduce delay. To fetch data for the FF subarrays, the data are first loaded from a Mem subarray to the global row buffer, and then they are written from the row buffer to the Buffer subarray. These must be done to avoid resource conflict i.e. global data line. when PRIME is accelerating NN computation, CPU can still access the memory and work in parallel as communication between the Buffer subarray and the FF subarrays is independent with the communication between the Mem subarray and the globe row buffer.

Prime controller decodes instructions and provides control signals to all the peripheral circuits in the FF sub array and configures FF subarray in memory and computation modes. Precision of input, output and synaptic weights effect our NN computation. 3-bit input and synaptic weight precision are adequate to provide 99% accuracy indicating NN algorithms are robust to precision bits. We use two 3-bit input signals to compose one 6-bit input signal and two 4-bit cells to represent one 8-bit synaptic weight.

PRIME programming involves three stages programming, compiling and code execution. It uses few APIs in programming stage like mapping the topology of NN to FF subarrays, program synaptic weights into mats, configure the data path of FF subarrays, running computation and post processing the results. The following figure 3.49 shows the high level workflow of in memory computation using PRIME.

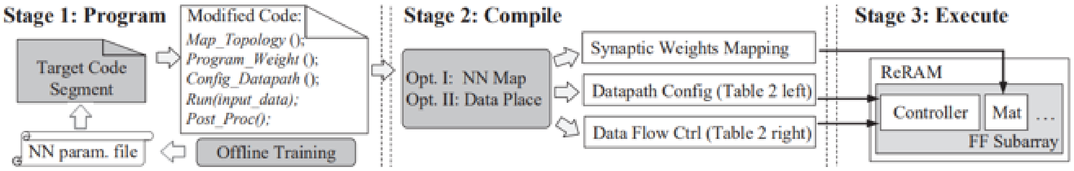


Figure 3.49. The software perspective of PRIME: from source code to execution

3.8.4 Evaluation

We compare PRIME with other counter parts. The benchmarks used have large NN and require high memory bandwidth. So, we can see enough improvements with PIM. PIM has 9.1X performance improvement compared to normal CPU as shown in figure 3.50. We see benefits because it doesn't have to map chips where data communication between banks is costly. Also, the weights are already preprogrammed in the cells. Prime also reduced memory access time to zero which means its access time is hidden by buffer arrays.

PIM architecture reduces memory access and saves energy. Further PRIME uses ReRAM cells which reduces the energy even more. The total energy consumptions are divided into three parts, computation energy, buffer energy, and memory energy. PRIME reduces energy for all

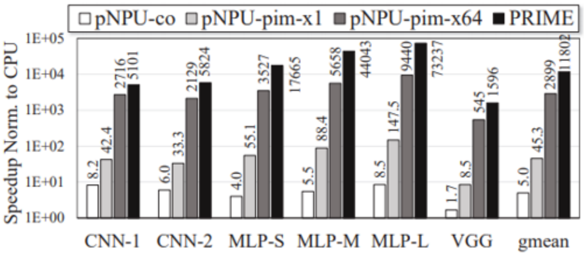


Figure 3.50. The performance speedups (vs. CPU).

the parts. Large storage size of ReRAM cells saves cache and memory access. Also, larger executions reduce buffer and memory access to temporary data. Given two FF subarrays and one Buffer subarray per bank (64 subarrays in total), PRIME only incurs 5.76% area overhead. The choice of the number of FF subarrays is a tradeoff between peak GOPS and area overhead. There is 60% area increase to support computation: the added driver takes 23%, the subtraction and sigmoid circuits take 29%, and the control, the multiplexer, and rest cost 8%.

3.8.5 Summary

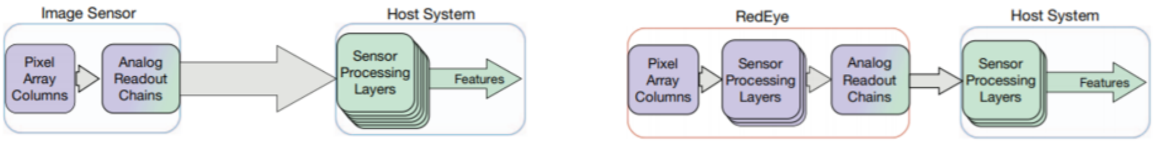
PRIME substantially improves the performance and energy efficiency for neural network (NN) applications, benefiting from both the PIM architecture and the efficiency of ReRAM-based NN computation. They can either perform computation to accelerate NN applications or serve as memory to provide a larger working memory space. With circuit reuse, PRIME incurs an insignificant area overhead to the original ReRAM chips. The experimental results show that PRIME can achieves a high speed up and significant energy saving for various NN applications.

3.9 Analog Computation

3.9.1 Introduction

Compared to digital circuits the energy efficiency of analog readout improves slowly with smaller transistors. Even though most circuitry moves to digital, we need analog to bridge the gap between physical world and digital realm. The main idea is to push the early processing to analog domain to reduce load on the analog readout. This is achieved by Red-Eye, an analog convolutional image sensor for continuous mobile vision. Red-Eye discards raw data, exporting features generated by processing a Conv-Net in the analog domain and hence saves energy.

Analog circuit face challenges with circuit design and porting to different technology node. Also, its processing accumulates noise. Red eye provides mechanism to tune noise parameters for efficiency. Red - Eye positions a convergence of analog circuit design, systems architecture, and machine learning, which allows it to perform early operations in the image sensor's analog domain, moving toward continuous mobile vision at ultra-low power. Figure 3.51 shows the transition that is done in this section to improve performance [24].



(a) Conventional sensor processing flow

(b) RedEye sensor processing flow

Figure 3.51. Conventional sensor processing incurs significant workload on the analog readout, early processing alleviates the quantization in the analog readout.

3.9.2 Overview

For image sensing, CMOS image sensors are used which consume hundreds of milliwatts. Modern image sensors consist of pixel array, analog readout and digital logic. The pixel array converts light into voltage signal, which are then modified by analog readout into digital values. Digital logic maintains sensor operation, managing timing generation, frame scanner and drivers, and data interface. Sensors have a column-based architecture where pixels in each column share a dedicated circuitry. The column amplifiers sample and amplify signals to avoid noise, but this consumes static and dynamic power. Digital optimizations like low power hardware for data reduction, early discard or computational offloading do not affect analog readout, which is the main reason for moving processing before analog readout. Analog computing is more efficient than digital computing. It uses single charge to represent a value reducing hardware count. Proper tradeoffs between energy efficiency and signal fidelity are made in analog domain. Maintaining state through capacitance C is susceptible to thermal noise. While raising C

$$\begin{aligned} \bar{V_n^2} &= kT/C \\ E \propto C &\propto 1/\bar{V_n^2} \end{aligned} \tag{3.7}$$

Deters noise, the energy required to fill the memory cell rises proportionally. Analog pipeline must be constructed in stages to facilitate configurability and maintainability which needs inter stage buffers. Arithmetic in charge-based circuits requires an operational amplifier which has

$$thermal \;\; noise \propto 1/c \quad \text{and} \quad energy \propto C \tag{3.8}$$

To readout we need significant energy. Successive approximation registers which trades accuracy for energy is used. SAR uses capacitors to approximate analog signal into digital signal which are the source of noise. Random errors in SAR are dominated by quantization noise. With ADC resolution n increased by each bit, quantization noise amplitude diminishes in half. However, increasing energy. Routing interconnects in analog is done via predefined routes to avoid congestion and overlap. Design complexity limits the portability between nodes. Red eye is designed to compute continuous mobile vision using CNN. Convolutional layer neurons multiply a three-dimensional receptive field of inputs with a kernel of weights to generate an output. A backpropagation process trains weights to alter the convolution operation. Other layers include nonlinearity and max pooling.

3.9.3 Architecture

Red eye is an analog computational image sensor architecture that captures an image and passes it to CNN while in analog domain while in analog domain as shown in figure 3.52. This reduces workload of analog to digital quantization and post sensor digital host system.

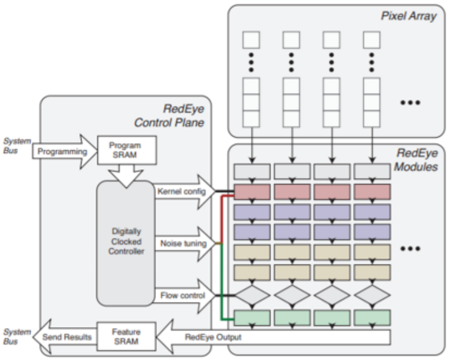


Figure 3.52. RedEye. The controller programs kernel weights,noise mechanisms, and flow control into RedEye modules.Output is stored in SRAM.

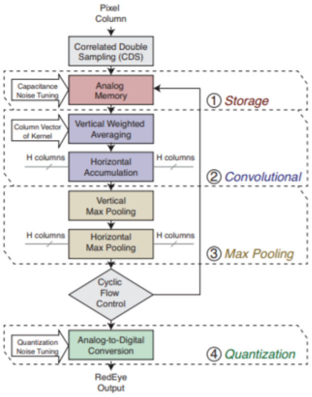


Figure 3.53. Convolutional column of the RedEye modules. A column of pixels is processed in a cyclic pipeline of convolutional and max pooling operations.

Leveraging that vision applications does not need high fidelity we move processing to analog domain. One challenge is that the noise gets accumulated which limits the number of stages, however red eye exploits the parallelism in CNNs. The architecture avoids redundant data by sending data through layers of processing modules as shown in figure 3.53, cyclically reusing modules before analog readout. As modules process on the patches of pixels, the complexity of interconnect is reduced by structuring the operation layers into a column-based topology. This arranges the modules in each processing layer in a column pipeline giving each processing module a physical proximity to input data. Red eye provides a conv net programming interface to directly load the programs into the SRAM. Also, a programmable noise admission mechanism is provided to tune for tradeoff between noise and energy saving.

The design exploits stackable structure of convnet to reuse red eye modules cyclically. Of the 4 modules the convolution and max pooling modules perform the operation of neuron in convnet layer. The storage module interfaces the captured or processed signal data with the processing flow and the quantization module exports the digital representation of the Red Eye Conv-Net result. The signal flow is controlled using synchronous digital clock. A set of control flow mechanisms are used to route signals for conv net processing. For additional processing, the cyclic flow control routes the output to storage module for intermediate storage. If not needed the bypass flow control of each module provides an alternative route to circumvent the module.

To promote data locality for kernel window operations, modules are organized in column parallel design which mitigates the complexity of unbounded interconnect structures. Using a clocked timestep we advance one row at a time allowing modules to work in parallel. In vertical direction data is buffered as generated, and for horizontal access, red eye bridge interconnects across horizontally proximate columns. This design pattern allows Red Eye to hierarchically reuse hardware design, thus enabling scalability. The Conv net program consisting of layer ordering layer dimensions, and convolutional kernel weights are loaded into SRAM. Then, digitally clocked controller is used to load program from SRAM into cyclic signal flow, issuing the specified kernel weights and noise parameters. Thus, a programmer should decide how much is processed on the analog and digital side. Noise admission mechanism is used to trade signal fidelity for energy efficiency. It uses the mechanisms to vary the capacitance of a damping circuit in operation modules which is configured during run time. Quantization noise is scaled by adjusting the ADC resolution of the analog readout.

A simulation framework is provided by taking in a developer's partitioned Conv Net and specified noise parameters. The structure of the framework maps to the sequential execution of modules. Simulation modifies a convnet framework to analyze the effects of noise. Two types of noise are injected into the processing flow. Gaussian noise layer models the noise inflicted by data transaction and computational operations. Red Eye uses the SNR to parametrize each Gaussian noise layer, allowing the developer to tune the noise in the simulation. The Quantization Noise Layer represents error introduced at the circuit output by truncating to finite ADC resolution. Red eye uses ADC resolution to tune the quantization noise layer introduced as shown in figure 3.54 and 3.55. Convolution is performed in two steps. Kernel weights are applied to signal values using tunable capacitors which are crucial for accuracy as shown in figure. Variable ADC resolution trades bit depth for energy efficiency. Significance of bit bi is

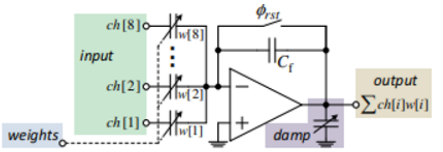


Figure 3.54. 8-input mixed-signal MAC. Tunable capacitors apply digital weights $w[:]$ on analog inputs $ch[:]$. ϕ_{rst} clears C_f after each kernel window is processed.

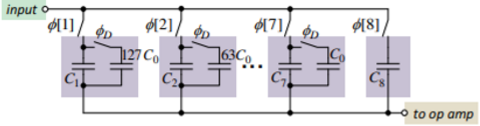


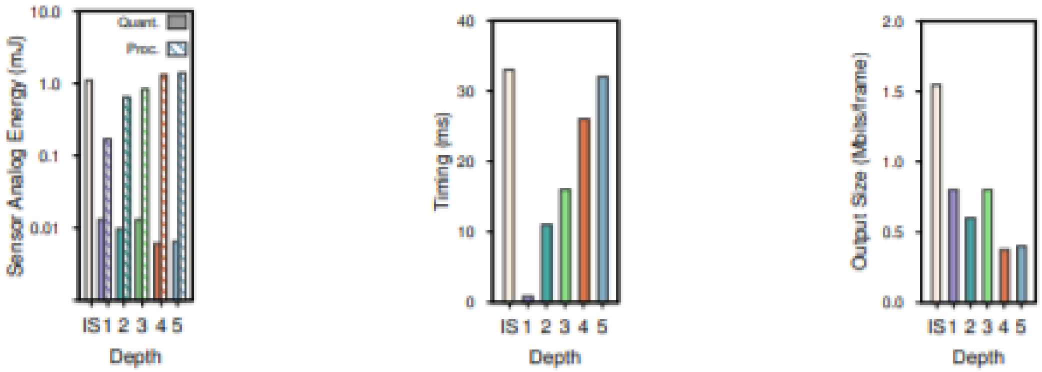
Figure 3.55. Tunable capacitor design. Digital weights $w[:]$ control the input-side switches $\phi[:]$ while sampling. C_1 to C_8 are identically sized to C_0.

defined by the weight of capacitor C_i w.r.t to total active capacitances of the n bit array.

$$wb_i = \frac{C_i}{C_\Sigma} = \frac{2^{i-1}C_0}{[\sum_{k=1}^{n} C_k] + C_0} = \frac{1}{2^{n-i+1}} \tag{3.9}$$

3.9.4 Evaluation

Top-level elements of convolution, max pooling and quantization layers are composed of sub-layer circuit modules, such as analog memory, MAC and ADC which are further dissected into internal units such as tunable capacitors, op amps and comparators. The energy consumption and noise contribution of each module is determined by set of noise, power and timing parameters. Noise parameters represents the noise performance of the circuit which is simulated using sample and hold circuit. For opamps we measure the output noise and refer it to input for consistency. For ADC the noise is identical to quantization noise of ideal m bit ADC. Power consumptions accounts for both static, bias current of opamps, leakage in digital circuits and dynamic power for comparators, tunable capacitors, digital elements when they flip. From figure 3.56 we find that Red Eye reduces analog sensor energy consumption by 84.5%. By pipelining Red Eye with computations on the digital system, the Depth5 Red Eye can operate at up to 30 frames per second (fps) consuming 170uj per frame. This noise reduction is achieved by through noise admission. Redeye computation can be offloaded to server; however, energy consumption is dominated by network transmission. But redeye reduces the output data transmission size reducing energy by 74%.



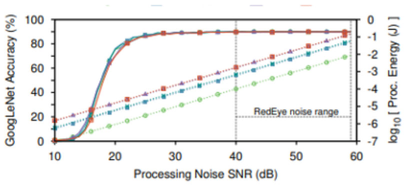
(a) Energy

(b) Timing

(c) Output size

Figure 3.56. Performance metrics of image sensor (IS) and 4-bit, 40 dB RedEye at different depths. Energy on a log scale.

In the below graph 3.57 and 3.58 gaussian noise is introduced to reduce SNR. The accuracy is 89% at low SNR limit of 40dB which could chosen for optimum efficiency.



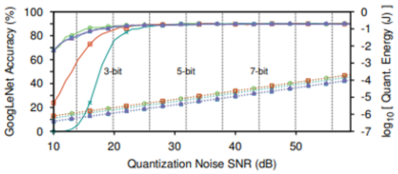


Figure 3.57. Accuracy and Energy of ConvNet processing vs. Gaussian SNR of GoogLeNet running on RedEye at 4-bit quantization.

Figure 3.58. Accuracy and Energy of Quantization vs. quantization SNR of GoogLeNet running on RedEye at Gaussian SNR = 40 dB.

On other hand while scanning ADC resolution at a fixed gaussian noise of 40dB, we find sizable accuracy trade off in the region of quantization scaling. With reduction in bits the accuracy reduces, however, for 4-6 bits all produces similar accuracy. However, with change in technology red eye faces challenges with technology scaling due to signal swing constraints, short channel effects and so on. Stacked Red Eyes could be programmed with different tasks to coexist on the same module and operate in parallel. Finally, conventional image processing architecture could occupy a layer, allowing a device to acquire a full image through Red Eye's optical focal plane when needed.

3.9.5 Summary

In this paper, we present the system and circuit designs of RedEye, moving ConvNet processing into an image sensor's analog domain to reduce analog readout and computational burden. With a perframe sensor energy reduction of 84.5%, cloudlet system energy reduction of 73%, and RedEye advances towards overcoming the energy-efficiency barrier to continuous mobile vision. Red eye noise enables operation till 30 dB and below with increased energy consumption. Processing such a ConvNet in the analog domain and discarding the raw image would provide a strong privacy guarantee to the user. Designing for convnet in digital domain has improved, however, we are trying to design for changes in analog domain which proves to have a good scope.

3.10 Multi Chip Scalability

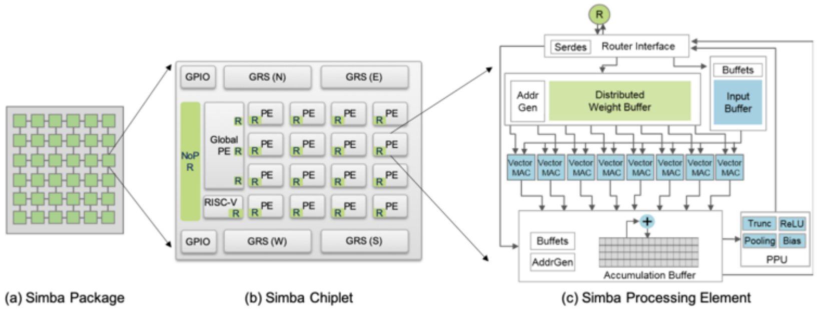
3.10.1 Introduction

Trend towards deeper networks, including compute and storage requirements motivate large-scale compute capability in DNN hardware, which is currently addressed by a combination of large monolithic chips and homogeneous multi-chip board designs. Multi-chip modules packaging approach can be used which reduces cost by employing smaller chiplet connected post-fabrication, as yield losses cause fabrication cost to grow super-linearly with die size. Recent advances in package-level signaling offer the necessary high-speed, high-bandwidth signaling needed for chiplet-based system. However, there are scalability issues due to the non-uniformity between on-chip and on-package bandwidth and latency.

In this section we discuss Simba, a scalable deep-learning inference accelerator employing multi-chip-module-based integration. Each of the Simba chiplets can be used as a standalone, edge-scale inference accelerator, while multiple Simba chiplets can be packaged together to deliver data-center-scale compute throughput. This focuses on non-uniform latency and bandwidth for on-chip and on-package communication that lead to significant latency variability across chiplets. In order to address this, a three tail-latency-aware, non-uniform tiling optimizations targeted at improving locality and minimizing inter-chiplet communication is used. Simba is the first work that highlights the challenge of mapping DNN layers to non-uniform, MCM-based DNN accelerators and proposes communication-aware tiling strategies to address the challenge.

3.10.2 Architecture

Tile-based architectures have frequently been proposed for deep learning accelerator designs. Design target is an accelerator scalable to data center inference. This is achieved by increasing the number of tiles in a monolithic single chip. However, building a flat network with hundreds of tiles would lead to high tile-to-tile communication latency. This is resolved by using a hierarchical interconnect to efficiently connect different processing elements using a network on chip and different chiplets on package using network-on-package.



(a) Simba Package (b) Simba Chiplet (c) Simba Processing Element

Figure 3.59. Simba architecture from package to processing element (PE).

Figure 3.59 shows a Simba package consisting of a 6 × 6 array of Simba chiplets connected via a mesh interconnect. Each Simba chiplet contains an array of PEs, a global PE, a NoP router, and a controller, all connected by a chiplet-level interconnect. To enable the design of a large-scale system, all communications are designed to be latency-insensitive and sent across the interconnection network through the NoC/NoP routers. The microarchitecture of the Simba PE includes a distributed weight buffer, an input buffer, parallel vector MAC units, an accumulation buffer, and a post-processing unit.

The heart of the Simba PE is an array of parallel vector multiply-and-add (MAC) units that are optimized for efficiency and flexibility. It uses a weight-stationary dataflow where weights remain in the vector MAC registers and are reused, while new inputs are read every cycle. To provide flexible tiling options, the Simba PE also supports cross-PE reduction with configurable producers and consumers. If current PE is the last PE on reduction chain, it sends partial sums to its local post-processing unit that performs ReLU, truncation and scaling, pooling, and bias addition. The final output activation is sent to the target Global PE for computation of the next layer. The Global PE serves as second-level storage for data to be processed by the PEs. The Global PE has a multicast manager that unicast data to one PE or multicast to multiple PEs for flexible partitioning. DNNs feature some computation that has low data reuse, such as elementwise multiply/add which can perform computations locally to reduce communication overhead for these types of operations and is called as near memory computation.

Each Simba chiplet contains a RISC-V processor core for configuring and managing the chiplet's PEs and Global PE states via memory-mapped registers using an AXI-based communication protocol. Synchronization of chiplet control processors across the package is implemented via memory-mapped interrupts. To efficiently execute different neural networks with diverse layer dimensions, Simba supports flexible communication patterns across the NoC and NoP. Both use a mesh topology with a hybrid wormhole flow control. Specifically, unicast packets use wormhole flow control for large packet size, while multicast packets are cut through to avoid wormhole deadlocks. Each Simba PE can unicast to any local or remote PE for cross-PE partial-sum reduction, to any local or remote Global PE to transmit output activation values, and to any local or remote chiplet controller to signal execution completion. A PE does not need to send multicast packets as its computation requires only point-to-point communication. Also, a Global PE can also send multicast packets to local and remote PEs for flexible data tiling.

Each Simba package as shown in figure 3.60 contains an array of 6×6 chiplets connected using ground-referenced signaling (GRS) technology for intra-package communication. The top and bottom rows of each chiplet include eight chipletto-chiplet GRS transceiver macros. Four macros are configured as receivers and four as transmitters. Each transceiver macro has four data

(a) Simba chiplet

(b) Simba package

Figure 3.60. Simba silicon prototype

lanes and a clock lane with configurable speed. We chose GRS as our communication mechanism because it delivers 3.5 higher bandwidth per unit area and lower energy per bit compared to other MCM interconnects. The prototype chiplets were implemented using a globally asynchronous, locally synchronous clocking methodology, allowing independent clock rates.

To map DNN layers onto the hierarchical tile-based architecture, we first use a state-of-the-art DNN tiling strategy that uniformly partitions weights spatially, leveraging model parallelism. As shown below each dimension of a DNN layer can be tiled temporally, spatially, or both at each level of the system hierarchy: package, chiplet, PE, and vector MAC. The compilation process starts with a mapper that is provided with data regarding available system resources and the parameters of a given layer from the Caffe specification. The mapper determines which PE will run each portion of the loop nest and in which buffers the activations and weights are stored. A random search algorithm is used to sample the mapping space and use the energy and performance models to select good mappings and placements. Finally, the flow generates the configuration binaries for each chiplet that implement the execution created by the mapper and placer.

3.10.3 Evaluation

Simba provides many mapping options, highlighting the importance of strategies for efficiently mapping DNNs to hardware. It demonstrates the highly variable behavior of different layers. The degree of data reuse highly influences the efficiency; layers with high reuse factors,

Hierarchical tile based structure

```
//Package level
for $p3$ = [0 : P3) :
    for $q3$ = [0 : Q3) :
        parallel_for $k3$ = [0 : K3) :
            parallel_for $c3$ = [0 : C3) :
// Chiplet level
for $p2$ = [0 : P2) :
    for $q2$ = [0 : Q2) :
        parallel_for $k2$ = [0 : K2) :
            parallel_for $c2$ = [0 : C2) :
// PE level
for r = [0 : R) :
    for $s$ = [0 : S) :
        for $k1$ = [0 : K1) :
            for $c1$ = [0 : C1) :
                for $p1$ = [0 : P1) :
                    for $q1$ = [0 : Q1) :
// Vector-MAC level
parallel_for k0 = [0 : K0) :
    parallel_for c0 = [0 : C0) :
        p = (p3 * P2 + p2) * P1 + p1;
        q = (q3 * Q2 + q2) * Q1 + q1;
        k = ((k3 * K2 + k2) * K1 + k1) * K0 + k0;
        c = ((c3 * C2 + c2) * C1 + c1) * C0 + c0;
        OA[p,q,k] += IA[p-1+r,q-1+s,c] * W[r,s,c,k];
```

tend to perform computation more efficiently than layers that require more data movement. Finally, although increasing the number of chiplets used in the system improves performance, it also leads to increased energy cost for chiplet-to chiplet communication and synchronization. Figure below shows a performance comparison of mapping ResNet-50's layer onto multiple PEs, either on-chiplet or spanning multiple chiplets. When mapped to a single chiplet, execution latency decreases linearly from one to eight PEs because of the improved compute throughput with more PEs as shown in figure 3.61.

At the same time, its performance flattens out beyond eight PEs due to the memory bandwidth contention at the Global PE's SRAM. However, when mapping across chiplets, execution time does not scale down beyond four PEs. The additional latency to communicate across multiple chiplets, including inter-chiplet communication latency and synchronization latency, ultimately leads to a 2 greater execution time relative to employing a single chiplet. Figure 3.62 shows the performance scalability of running three different layers in ResNet-50 across different numbers of chiplets. While performance improves for all the layers till 8 after that is hindered by the

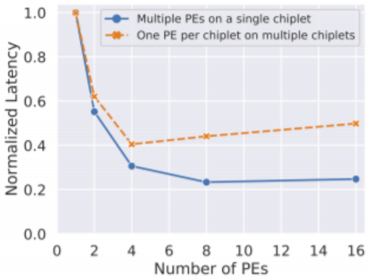


Figure 3.61. Performance comparison of on-chip and onpackage communication and synchronization in Simba.

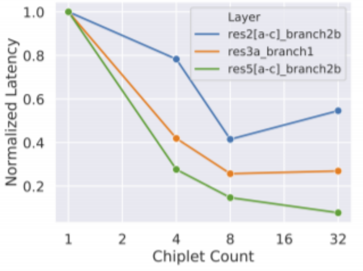


Figure 3.62. Simba scalability across different layers from ResNet-50.

communication overhead and the amount of weights used to be fully utilized by Simba. This shows the amount of compute parallelism that an MCM can leverage varies from layer to layer, and that the cost of communication can hinder the ability to exploit that parallelism, even on a single chiplet.

To examine the bandwidth sensitivity of different layers, we adjust the bandwidth of the NoP relative to the intra-chiplet compute performance of the system. This adjustment is made by reducing the frequencies of the PE, Global PE, and RISC-V partitions below nominal while maintaining a constant NoP frequency. The figure 3.63 shows that execution time is affected by NoP bandwidth for two layers. For different layers increasing bandwidth reduces the execution time accordingly. Because an MCM-based system intrinsically has a NUNA architecture between intra-chiplet and inter-chiplet PEs, mapping policies must consider the different latency and bandwidth parameters to deliver good performance and efficiency.

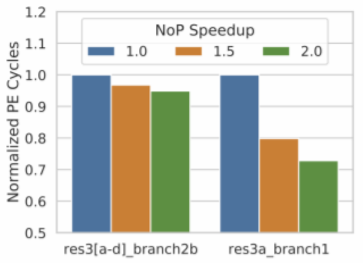


Figure 3.63. Simba scalability with different chiplet-to-chiplet communication bandwidths.

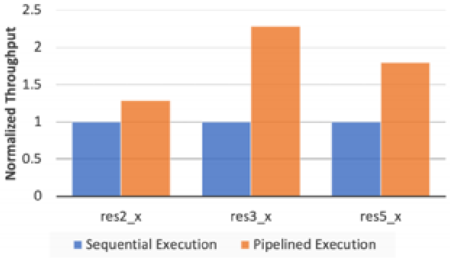
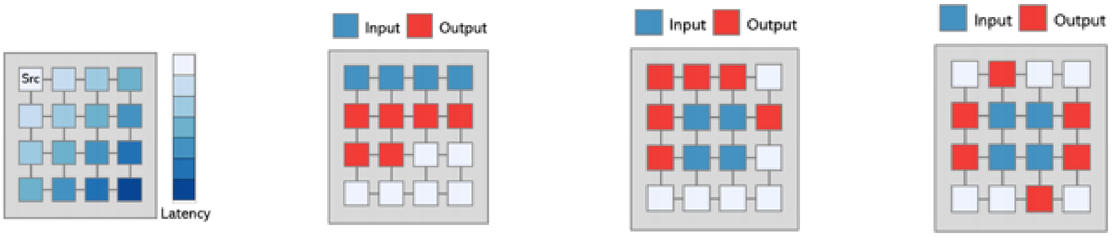


Figure 3.64. Throughput improvement from pipelined execution on 3 blocks of ResNet-50.

In addition to lower bandwidth, the NoP has higher latency than the NoC due to inter-chiplet signaling overheads. To isolate the effect of NoP latency, we ran experiments mapping layers to four chiplets, but adjusted the locations of the selected chiplets in the package to modulate latency. With active chiplets further apart as shown in figure 3.64, the overall execution time increases by up to 2.5 compared to execution on adjacent chiplets. Communication latency is typically less pronounced for small-scale systems but plays a significant role in achieving good performance and energy efficiency for a large-scale, MCM-based system like Simba.



(a) Latency Profile (b) Initial Placement (c) Input Placement (d) Output Placement

Figure 3.65. Data placement on the Simba system.

DNN workload tiling techniques that target the non-uniform latency and bandwidth is presented which focusses on importance of communication-latency-aware tiling when mapping DNN workloads to large-scale, hierarchical systems. Large scale systems with PEs spatially distributed have varied latencies. This is handled by non-uniformly partition the work across the PEs as shown in figure 3.65. PEs closer to the data producers will perform more work to maximize physical data locality, while PEs that are further away will do less work to decrease the tail latency effects. Below figure 3.66 shows that different PEs have different data arrival times. Greedy algorithm to determine where input and output activation data should be placed in the Simba system. Pipelined execution on Simba improves overall throughput by up to 2.3 compared to the sequential execution baseline.

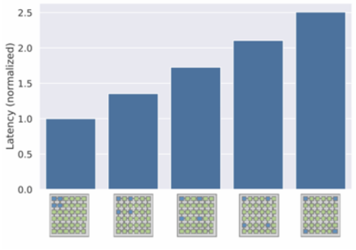


Figure 3.66. Simba scalability with different chiplet-to-chiplet communication latencies.

3.10.4 Summary

This work presents Simba, a scalable MCM-based deep-learning inference accelerator architecture. Simba is a heterogeneous tile-based architecture with a hierarchical interconnect. A silicon prototype system consisting of 36 chiplets that achieves up to 128 TOPS at high energy efficiency. The prototype to characterize the overheads of the non-uniform network of an MCM-based architecture is used, observing that load imbalance and communication latencies contribute to noticeable tail-latency effects. The non-uniform nature of system can help improve performance through techniques such as nonuniform work partitioning, communication-aware data placement, and cross-layer pipelining. Applying these optimizations results in performance increases of up to 16% compared to naive mappings.[18][12][10][11]

3.11 Tensor Processing Unit

3.11.1 Introduction

Deep neural networks have led to break through in many aspects for wide range of targets like speech, vision, language translation, search and many more. These have brain like functionality based on a simple artificial neuron which is a nonlinear function of a weighted sum of inputs. These neurons are arranged in layers with output of one layer going to input of another layer. Layers are made deeper by increasing them to improve accuracy in some cases. The two phases of neural network are training, and inference referred to as development versus production. The developer decides the number of layers and type of network while training determine weights.

Training is done in floating point which made GPU popular as training is compute intensive. However, using quantization transforms floating point into fixed point like 8 bit which is good for inference. Neural networks like multi-layer perceptron, convolutional and recurrent neural networks are popular. Training weights consumes energy and can be amortized by reusing across batch of independent examples [28].

3.11.2 Architecture

Few applications need excess capacity on data centers, which can be improved using specialized hardware. Goal was to improve the performance by 10X compared to GPU. To reduce the dependency with CPU, TPU was designed as a co-processor on the PCIe I/O bus.

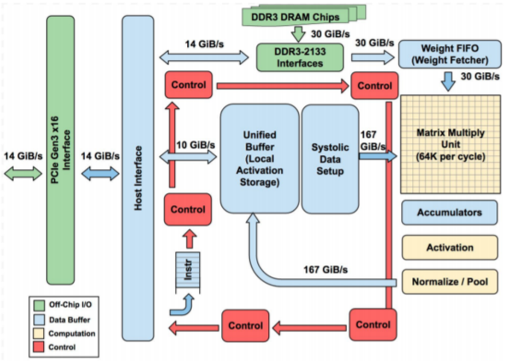


Figure 3.67. TPU Block Diagram

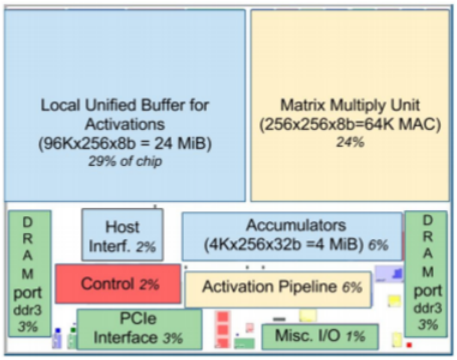

Local Unified Buffer for Activations (96Kx256x8b = 24 MiB) 29% of chip
Matrix Multiply Unit (256x256x8b=64K MAC) 24%
DRAM port ddr3 3%
Host Interf. 2%
Accumulators (4Kx256x32b =4 MiB) 6%
DRAM port ddr3 3%
Control 2%
Activation Pipeline 6%
PCIe Interface 3%
Misc. I/O 1%

Figure 3.68. Floor Plan of TPU die

The goal is to run inference models in TPUs to reduce interaction with hosts and flexible enough for different networks. The TPU instructions are sent from the host over PCIe into instruction buffer and the internal blocks. As shown in figure 3.67 Matrix multiply is heart of TPU, contains 256256 MAC and results are accumulated on an accumulator. The weights for the matrix come from an on-chip weight FIFO that reads values from off-chip DRAM called weight memory which is four tiles deep. A programmable DMA controller transfers host memory to and from CPU and unified buffer. Figure 3.68 shows the floor plan of a TPU. Instructions are sent on relatively slow PCIe bus and TPU instructions follow CISC instruction. Input flows from right to left and partial sums flow from top bottom resembling a waterfall structure as shown in figure 3.69. This operation takes B*256 input multiplies it 256*256 constant weight input to produce B*256 output, taking B pipeline cycles to complete.

The TPU architecture keep the matrix unit busy by using a 4-stage pipeline for this CISC instruction. The other instructions are overshadowed by overlapping their execution with matrix multiply instruction. The pipeline does not overlap properly as the CISC instructions can occupy a station for thousands of clock cycles in contrast to RISC pipeline with one clock cycle per stage. In cases when activations for one network should complete before matrix multiplication for the next layer can begin, a delay slot is seen and needs to wait for explicit synchronization.

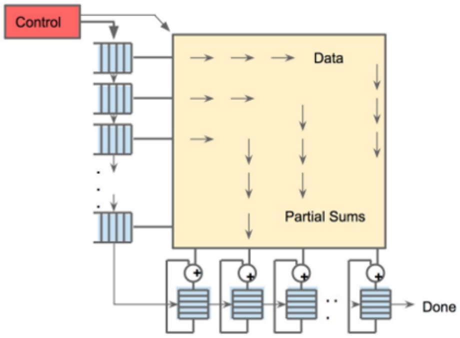


Figure 3.69. Systolic data flow of the Matrix Multiply Unit.

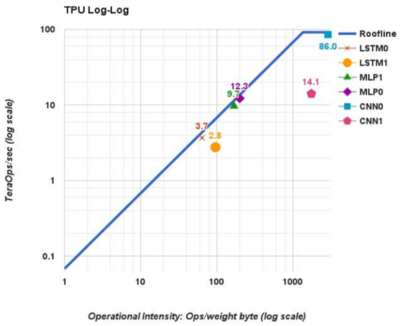


Figure 3.70. TPU (die) roofline.

Reading from SRAM consumes more energy than arithmetic, the matrix unit uses systolic execution to save energy by reducing reads and writes of the Unified Buffer. The data comes from different directions arriving at cells in an array at regular intervals where they are combined. A given 256 element MAC operation moves through matrix as a diagonal wave front with weights loaded from top and activations flowing in from left to right. The weights are preloaded and start taking effect with the first block of data. Control and data are pipelined to give the illusion that the 256 inputs are read at once, and that they instantly update one location of each of 256 accumulators. The TPU software stack should be compatible with those developed for CPU, GPU for portability. Like GPUs, TPU stack is split up user space driver and kernel driver. Kernel driver is designed for long term stability while the user space driver sets up and controls TPU execution, reformats data into TPU order, translates API calls into TPU instructions. The user space compiles a model first time it is evaluated, caching the program image, and consecutive runs are done at full speed. Computation is often done one layer at a time, with overlapped execution allowing the matrix multiply unit to hide most non-critical-path operations.

3.11.3 Evaluation

Roofline performance model as shown in figure 3.70 is used which assumes that applications do not fit in on chip caches and so they are computation limited or memory bandwidth limited. For high performance computing we measure the floating-point operations per second with operational intensity Without enough operational intensity program is memory bandwidth-bound and lives under the slanted part of the roofline.

Table 3.5. Factors limiting TPU performance of the NN workload based on hardware performance counters.

Application	MLP0	MLP1	LSTM0	LSTM1	CNN0	CNN1	Mean	Row
Array active cycles	12.7%	10.6%	8.2%	10.5%	78.2%	46.2%	28%	1
Useful MACs in 64K matrix(%peaks)	12.5%	9.4%	8.2%	6.3%	78.2%	22.5%	23%	2
Unused MACs	0.3%	1.2%	0.0%	4.2%	.0%	23.7%	5%	3
Weight shal cycles	53.9%	44.2%	58.1%	62.1%	0.0%	28.1%	43%	4
Weight shift cycles	15.9%	13.4%	15.8%	17.1%	0.0%	7.0%	12%	5
Non-matrix cycles	17.5%	31.9%	17.9%	10.3%	21.8%	18.7%	20%	6
RAW stalls	3.3%	8.4%	14.6%	10.6%	3.5%	22.8%	11%	7
Input data stalls	6.1%	8.8%	5.1%	2.4%	3.4%	0.6%	4%	8
TeraOps/sec(92 Peak)	12.3%	9.7%	3.7%	2.8%	86.0%	14.1%	21.4%	9

Table 3.6. Time for host CPU to interact with the TPU expressed as percent of TPU execution time.

MLP0	MLP1	LSTM0	LSTM1	CNN0	CNN1
21%	76%	11%	20%	51%	14%

The gap between the actual operations per second and the ceiling shows potential benefit of further performance tuning. For using the roofline for TPU the NN applications are quantized by replacing floating point operations with integer operations. We redefine operational intensity as integer operations per byte of weights read because weights do not normally fit in on chip memory. The slanted line in the above figure depicts the applications are memory bandwidth limited than compute. So, MLPs and LSTMs are memory bound where as CNNs are compute bound. CNN1 despite higher operational intensity runs at 14 TOPs compared to CNN0 which is 86 TOPs. TPU spends less than half of its cycles performing matrix operations for CNN1 and of those active cycles, only about half of the 65,536 MACs hold useful weights because some layers in CNN1 have shallow feature depths. About 35% of cycles are spent waiting for weights to load from memory into matrix unit, which occurs during the 4 fully connected layers that run at an operational intensity of just 32. Having said that we cannot account for the exact cycles because of the overlap of executions on the TPU.

The following table 3.5 and 3.6 shows TPU performance. Queuing theory shows that long queues raise the throughput but stretches the response time as instructions are waiting in the queue. So ideally the queues are made sure to be empty. Comparisons suggests that GPU die is 1.1X than CPU and TPU is 14.5X and thus TPU is 13.2 compared to GPU. We use geometric mean when we do not know the composition, however using weighted mean we have TPU to be 15.3 compared to GPU performance. The best cost metric in a datacenter is total cost of ownership (TCO). This can not be published as this information helps to deduce. However, performance/watt results are publishable. In the below figure 3.71 we see the performance relative to CPU. The first two include the host CPU server for GPU and TPU and later two subtracts the host CPU power. The TPU server has 14 to 16 times the performance/Watt of the GPU server. Thermal design power affects cost as we must provide enough cooling and power when at full power. Also, servers are 100% busy less than 10% of the time and advocated energy proportionality i.e. servers should consume power proportional to the amount of work performed. Though TPUs consume less power there, there energy proportionality is poor. At 10% load, the TPU uses 88% of the power it uses at 100% as the short design schedule prevents inclusion of many energy saving features.

There are parameters with which the TPU performance is sensitive to. Increasing memory bandwidth by 4X improves performance by 3X. Clock rate has little benefit on average with or without more accumulators as only CNNs are compute bound. 3X improvement in clock rate increases the performance by 2X. The performance degrades as the matrix unit expands. Even

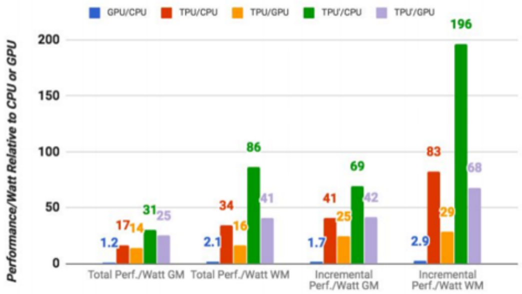


Figure 3.71. Relative performance/Watt (TDP) of GPU server and TPU server to CPU server, and TPU server to GPU server.

tough we can reduce steps by tiling larger units each step takes 4X ore time.

3.11.4 Summary

In summary TPU succeed because of the large matrix multiply units, software controlled on-chip memory, ability to run whole inference and reducing the dependency on CPU. This single threaded deterministic model has enough flexibility to work on different models. Omission of general-purpose features consumes less energy and area despite the larger data path and memory. Quantization of applications to use 8-bit integers and that applications were written using TensorFlow, which made it easy to port them to the TPU at high-performance rather than them having to be rewritten to run well on the very different TPU hardware. Order-of-magnitude differences between commercial products are rare in computer architecture, which may lead to the TPU becoming an archetype for domain-specific architectures.

3.12 Real Time AI

3.12.1 Introduction

Hardware acceleration of DNNs is becoming commonplace and the high level of parallelism available makes them amenable to acceleration. Training and inference are different phases needed for acceleration. While training is throughput bound, Inference is more latency sensitive. Highly parallel architectures with deep pipelines, achieve high throughput on DNN models by batching evaluations, exploiting parallelism both within and across requests. This approach works well for offline training, where the training data set can be partitioned into minibatches, increasing throughput while not impacting convergence. In an online inference, requests often arrive one at a time. Requests must be processed individually, leading to reduced throughput while still sustaining batch-equivalent latency, or incur increased latency by waiting for multiple request arrivals to form a batch[31].

Rather than increasing throughput by exploiting inter request parallelism, the system reduces latency by extracting as much parallelism as possible from individual requests. Real-time AI is used to describe DNN inference with no batching and is performed by Brainwave. Despite the lower clock rate and higher area overheads that FPGAs incur, the BW NPU performs better for real-time AI, sustaining 35 Teraflops on large RNN benchmarks with no batching. BW NPU achieves low latency on individual DNN requests using reconfigurable logic, and a hard NPU with a higher clock rate but reduced flexibility. Since the BW NPU provides a high-level single-threaded abstraction, the underlying microarchitecture can vary widely allowing flexibility in implementation. The BW NPU accepts four synthesis-time parameters that can optimize the microarchitecture resources. These are data type, native vector size, number of data lanes, and size of the matrix-vector tile engine. Finally, BW achieves high throughput without sacrificing low latencies.

3.12.2 Background

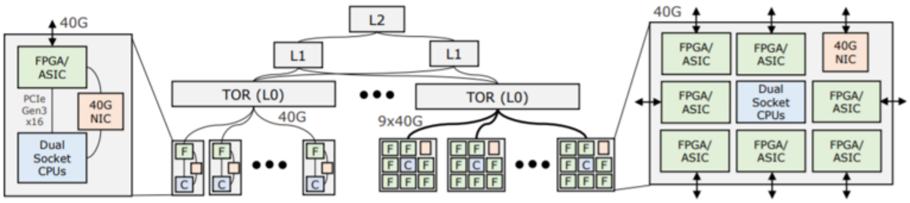
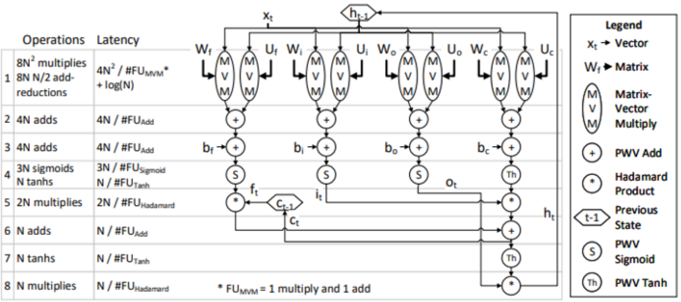


Figure 3.72. BW system at cloud scale.

The BW system as shown in figure 3.72 integrates with an existing hyperscale cloud infrastructure that runs production services with real-time AI requirements. Figure illustrates components of a hyperscale datacenter. Every standard dual-socket server hosts PCIe-attached accelerator cards that contain FPGAs or ASICs. The accelerator cards have direct access to the datacenter network and are placed in-line between the server NIC and the top-of-rack switches. The accelerators communicate directly using an on-chip RDMA-like lossless protocol. The datacenter architecture can be logically disaggregated and pooled into instances of hardware microservices. Once initialized and registered with a distributed resource manager, a given hardware microservice is published to subscribing CPUs in the system and accessed directly through an IP address. The CPU and FPGA resources devoted to acceleration are scaled independently. Large, partitionable problems can be spatially distributed across multiple accelerators

The DNN accelerator consists of three components: a tool flow that transforms pre-trained DNN model checkpoints into software and accelerator executables, a federated runtime that orchestrates model execution between CPUs and distributed hardware microservices, and the programmable BW NPU instantiated on FPGAs. In the initial phases of the tool flow, a pre-trained DNN model is exported from a DNN framework into BW's graph intermediate representation. The GIR undergoes a series of optimizations and transformations based on target constraints of the backend system. In real-time scenarios, the tool flow can partition large graphs that exceed the capacity into sub-graphs whose parameters can be pinned individually into accelerators on chip memory removing the hindrance of performance. Operations not supported on BW are executed on CPU.

Operations per second and energy efficiency do not capture the effects of batching, which can artificially drive up utilization while increasing latency. So, latency-centric metrics are introduced based on critical path analysis. These metrics enable robust NPU evaluation by characterizing the latency gap from idealized implementations. UDM reflects the lower bound latency capturing all available parallelism of a single DNN request; whereas SDM reflects the lowest possible latency under realistic resource constraints and assesses how well an implementation exploits available parallelism of a single DNN request with high microarchitectural efficiency. Figure



	Operations	Latency
1	$8N^2$ multiplies 8N N/2 add-reductions	$4N^2$ / $\#FU_{MVM}$* + log(N)
2	4N adds	4N / $\#FU_{Add}$
3	4N adds	4N / $\#FU_{Add}$
4	3N sigmoids N tanhs	3N / $\#FU_{Sigmoid}$ N / $\#FU_{Tanh}$
5	2N multiplies	2N / $\#FU_{Hadamard}$
6	N adds	N / $\#FU_{Add}$
7	N tanhs	N / $\#FU_{Tanh}$
8	N multiplies	N / $\#FU_{Hadamard}$

Figure 3.73. LSTM critical-path analysis. Operation count and latency are shown as functions of LSTM dimension (N) and number of functional units (FU)

3.73 illustrates a critical-path analysis applied to the dataflow of a long short-term memory block. The LSTM requires 64M operations per time step and can be executed in 19 cycles on an idealized UDM. The more realistic SDM constrained to 96,000 multiply-accumulators and serves the model in 352 cycles. The 18X gap between these metrics suggests that performance improvements are obtained with more resources. LSTMs exhibit lower parallelism and higher data requirements making it challenging to accelerate. Microarchitectural inefficiencies such as data and structural hazards, pipeline stalls, and limited memory bandwidth conspire to prevent NPU implementations from approaching ideal latencies.

3.12.3 Architecture

BW NPU architecture provides a simple programming abstraction that can be targeted easily by programmers and compilers, encoding sufficient information of large DNN operations that an underlying microarchitecture can efficiently exploit parallelism, and supports flexibility. BW NPU adopts a single-threaded SIMD ISA made up of compound operations that operate on $1-D$ and $2-D$ fixed-size vectors and matrices. Sub graphs are encoded through atomic instruction chain which captures communications between graph edges. It exposes specialized instructions, datatypes, and memory abstractions that are optimized for low-latency DNN serving. The BW NPU matrix/vector data path is implemented as a coprocessor, using a conventional scalar core to issue BW NPU instructions to the data path via an instruction queue. This provides the BW

NPU's control flow, including dynamic input-dependent control flow, a critical requirement for certain models such as single-batch RNNs with variable length timesteps.

Basic linear algebra routines can be performed at three canonical levels: vector-only, Level, matrix vector, matrix-matrix operations. The BW NPU architecture focuses on matrix-vector multiplication as its key operation.

The BW NPU instruction set accommodates DNN models spanning LSTMs, GRUs, 1D, 2D CNNs, dense MLPs. In BW NPU, all instructions operate on N-length 1D vectors or NN 2D matrices. The read and write operations (v_rd, v_wr, m_rd, m_wr) use their first operand to select a memory target, which could be a specific register file, DRAM, or a network I/O queue and second operand provides a memory index, except in the case of network I/O. While other instructions, access memory structure identified by opcode where it needs an index operand. Matrices can be read only from the network or from DRAM and can be written only to the matrix register file or to DRAM. The MRF is read only as an implicit operand of a matrix-vector multiply(mv_{mul}). BW NPU ISA has explicit instruction chaining, which allows the microarchitecture to exploit pipeline parallelism without complex hardware dependency checking or multi-ported register files. The BW NPU enables operation on multiples of the native dimension using by setting scalar control registers using s_{wr}. This capability has been used with great extent to parameterize models, with the added benefit of reducing instruction bottlenecks. BW NPU microarchitecture goal is to exploit the vector-level parallelism of a single DNN request at high hardware efficiency. In practice, pipeline stalls caused by hazards, decoding inefficiencies, and inherent serial data dependences in models limit the exploitable VLP within a single request.

Figure 3.74 shows a high-level view of the BW NPU microarchitecture. The primary goal is to map and execute instruction chains to a continuous, uninterrupted stream of vector elements flowing through the function units. The function units form a linear pipeline, mirroring the instruction chain structure, with the matrix-vector multiplier at the head. The vector arbitration network manages data movement among the memory components: pipeline register files, DRAM, and network I/O queues. A top-level scheduler configures the control signals for the function

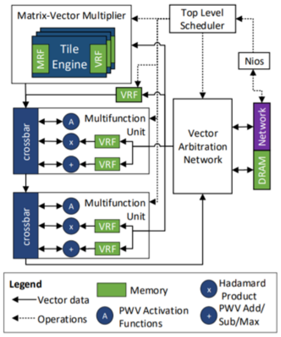


Figure 3.74. Microarchitecture overview

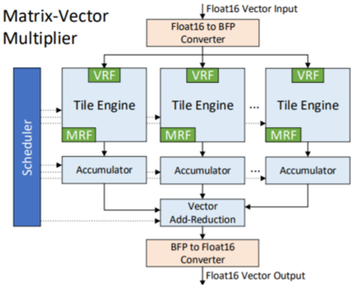


Figure 3.75. Matrix-vector multiplier overview.

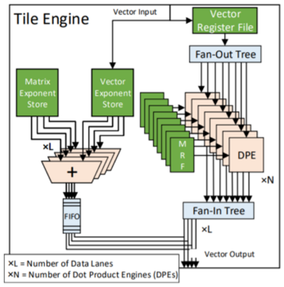


Figure 3.76. Matrix-vector tile engine microarchitecture

units and vector arbitration network based on the BW ISA instruction chains it receives from the scalar control processor.

The matrix-vector multiplier in figure 3.75 is the primary workhorse of the BW NPU and is scaled across a network of dot product unit and is memory-bandwidth limited. To alleviate this bottleneck, each input to every single dot product unit requires a dedicated memory port to feed the units at maximum throughput. A hierarchical view of the MVM is depicted. At the highest level, it instantiates a series of matrix-vector tile engines, each of which implements a native-sized MVM. In turn, each tile engine in figure 3.76 is made up of a series of dot product engines, as shown in next Figure. Each dot product engine is responsible for the dot product computation that corresponds to multiplying the input vector by one native row in the matrix tile. The dot product engines are composed of lanes of parallel multipliers that feed into an accumulation tree. These lanes provide parallelism within the columns of a row of a matrix tile. Combined, the MVM exploits four dimensions of parallelism: inter-MVM, MVM tiling, across the rows of a tile, and within the columns of the row. The total throughput of an MVM in FLOPs per cycle can be expressed as $2 \times \#tile\ \ engines \times \#DPEs \times \#lanes$. Each multiplier receives one vector element and one matrix element per cycle. The matrix elements are delivered by a dedicated port of the matrix register file positioned adjacent to that multiplier. The MRF is banked by native tiles across the tile engines. Each bank is further sub-banked by rows, such that the first sub-bank in a tile engine contains the first row of every matrix tile in the MRF bank. The elements of each row are interleaved in SRAM such that each SRAM read port can be directly connected to a multiplier. This organization scales the number MRF read ports with local compute tiles. MRF entries can be written only from DRAM or the network input queue, so write bandwidth requirements are much lower. Because MVM performance depends on MRF bandwidth, needed matrix operands must fit in the available on-chip SRAM to achieve high utilization. The output from the MVM is routed through a series of vector multifunction units. The MFUs support vector-vector operations such as multiplication and addition as well as unary vector activation functions like ReLU, sigmoid, and tanh. Dedicated vector register files associated with the add/subtract and multiply function units provide the secondary operands needed for those operations. Each MFU functions units and IO ports are connected by non-blocking crossbars. The crossbar is configured according to the current instruction chain to route data. Once configured, a sequence of vectors can be pipelined through the MFU. The BW NPU uses a conventional scalar control processor to dynamically issue BW NPU instructions asynchronously to the top-level scheduler. The top-level scheduler must decode each instruction chain into thousands of primitive operations to control the operation of many spatially distributed compute resources. The hierarchical decode and dispatch logic expands compound operations into distributed control signals that manage compute units, register files and switches.

The top-level scheduler dispatches to 6 decoders and 4 second-level schedulers, which in turn dispatch to an additional 41 decoders. This scheme, combined with buffering at each stage, keeps the entire

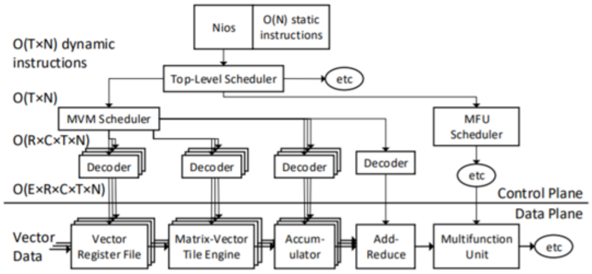


Figure 3.77. Hierarchical decode and dispatch (HDD) into the matrix-vector multiplier

compute pipeline running with an average of one compound instruction dispatched from the Nios every four clock cycles. Figure 3.77 illustrates how the largest functional unit is controlled from a single instruction. An expansion of decoding information occurs from top-to-bottom as the Nios processor streams T iterations of N static instructions into the top-level scheduler. Next, the top-level scheduler dispatches the MVM-specific portion of instructions to a second-level scheduler, which expands operations along the target matrix's R rows and C columns . These MVM schedules are mapped to E matrix-vector tile engines, with operations dispatched to a set of E decoders each for the tile engines and their associated vector register files and accumulation units, along with a monolithic add-reduction unit. Finally, these decoders generate control signals that fan out into the data plane, with each tile engine dispatcher fanning out to hundreds of dot product engines. The BW microarchitecture can be viewed as a fully parameterizable processor family that can be customized to specific models for efficiency.

The BW architecture exposes several major parameters that can be used for specializing a micro architecture instance to specific models: aligning the native vector dimension to parameters of the model tends to minimize padding and waste during model evaluation, increasing lane widths can be used to drive up intra row-level parallelism, increasing matrix multiply tiles can exploit sub-matrix parallelism for large models.

3.12.4 Evaluation

Batch size of 1 provides lowest cloud latency as requests are processed immediately and it simplifies everything as batching queues are not needed. The BW NPU can run all Deep Bench layers at under 4ms at batch 1, reaching up to 35.9 effective TFLOPS for a large GRU over hundreds of timesteps providing two orders of magnitude advantage over GPU. This happens because of high hardware utilization.

The figure 3.78 and 3.79 shows the utilization, which is the percentage of the peak TFLOPS reached for each layer. At batch size of 1, the BW NPU reaches 23% to 75% of peak FLOPS for medium to large LSTM/GRUs. There is 4-23X improvement as BW NPU can fully expose on-

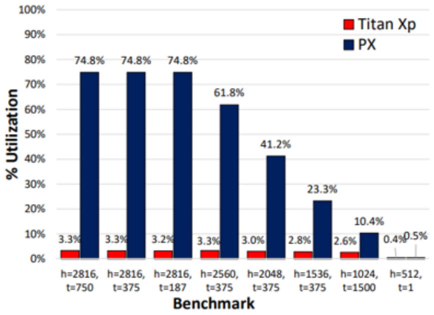


Figure 3.78. Hardware utilization across DeepBench RNN inference experiments.

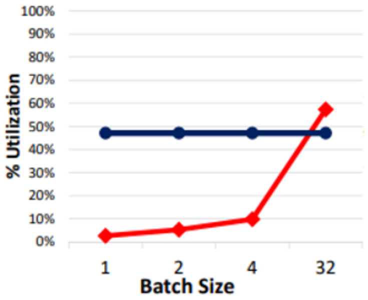


Figure 3.79. Utilization scaling with increasing batch sizes(LSTM-2048).

chip RAM bandwidth, pipeline dependent RNN vectors and exploit all degrees of matrix-vector parallelism to keep its compute units busy. As hidden dimension reduces the compute utilization reduces due to the large native tile dimension, which can result in wasteful work and the deep pipelines that delays dependent data from being written back quickly. However, the BW can be reconfigured to different degrees of parallelism to recover metrics.

When SDM latencies are compared to BW, BW is within a factor of 2.17X for the large GRUs and LSTMs. However, this factor falls off for the remaining models because the high dimension MVMs and deep pipeline of the BW lead to essentially the same latency per time step in steady state for all evaluated models regardless of their size, between 252 and 296 microseconds. For cloud services which can tolerate more latencies small amount of batching is employed. The second graph shows the utilization scaling with increasing batch size. Since BW executes a single input at a time, with increased batch size, the utilization remains constant. In contrast, GPU utilization increases proportionally as batch size increases since there is now more independent parallel work to fill the GPU SMs. However, at batch size of 4, the GPU remains at under 13% utilization even for large RNNs. Even though 32 improves the utilization, such large batch sizes do not exist in real life. The power was measured when all the Ips are being used which turned out to be 125W which provides efficiency of BW at 287 GFLOPS/W when running large models at high device utilization.

3.12.5 Summary

Brain Wave achieves high throughput and low latency for real-time AI, with no mini batching. The system pins models in on-chip memories and extracts mega-SIMD parallelism from a single thread of control, with some of the compound instructions generating millions of independent operations. Hierarchical decode and dispatch breaks these operations into fleets of fixed-length vector operations that are then scheduled on a distributed substrate, operating in parallel and exploiting direct producer-consumer dataflow routing to reduce pipeline bubbles. Taken together, these techniques allow higher utilization and lower latency on a collection of RNNs. For the larger models, the latencies are 10-90X lower than the GPU, and effective utilization is higher than the GPU for all benchmarks until a batch size of 32 is applied. As the resources grow, so must the native vector length, so control overheads do not start to dominate. As the frequency is pushed, performance will grow but efficiencies will drop with increased pipeline bubbles. Like CPUs exploiting ILP, the NPU space must find the best balance of frequency and efficiency for exploiting vector-level processing, which is currently unknown.

3.13 Conclusion

Accelerator design has come a long way. The need for higher speed and efficiency in DNN execution has led to explosion of DNN accelerators. In section 3.2 basic design of an accelerator is discussed. Section 3.3 covers different dataflow taxonomy. Optimized dataflow helps reduce the amount of data transfer from on chip to main memory, reducing power. With specialized hardware using more granular instruction set architecture helps to make the workflow efficient. A novel ISA is ducussed which provides better performance than the high level instructions. Data movement as mentioned before is crucial for energy savings and having an optimized network on chip is needed for better performance. Lot research has been in place to improve the compute part of hardware accelerator and data flow taxonomy. A new perspective would be to reduce the scope for computation itself. In 3.7 and 3.8 we discuss about variable bitwidth optimization and compressed sparse neural networks which are used to reduce the amount of computation with out loss of accuracy. A more recent approach of computing inside main memory has revolutionised the performance of compute and data movement in hardware accelerator domain. Analog computation can also be used to reduce the power as these circuitry require less energy than the digital counterparts.Real time AI issues has been resolved using Brainwave. Finally all this modifications have been proposed for a single chip. However, multi chip modules can be used to increase the throughput without increasing exponential power consumption and unbalanced bandwidths. However multi tenancy has been largely omitted in proposed or deployed designs due to arms race in the market for higher speed and efficiency. Even the MLPerf benchmark suit keeps this single-model focus for both training and inference. In contrast, we discuss in the next section Planaria, which offers spatial multi-tenant acceleration through architecture fission that is propelled by unique microarchitectural mechanisms and organizations that enables flexible task scheduling. As such, this paper lies at the intersection of DNN acceleration and multi-tenant execution.

Chapter 4

Planaria: Dynamic Architecture Fission for Spatial Multi-Tenant Acceleration of Deep Neural Networks

4.1 Introduction

Computer industry took advantage of Moore's law and Dennard scaling for several decades and significantly improved the performance of single core processors. Following Moore's law, the number of transistors on chip has grown exponentially for decades. This growing transistor count, coupled with recent architecture and compiler advances, has resulted in an unprecedented exponential performance increase of computers. However, with the end of Dennard scaling which argued that one could continue to decrease the transistor feature size and voltage while keeping the power density constant, has raised a big challenge for large transistor count IC designs. Power and energy became the major constraints of digital systems. To offer higher performance without significant increase in power, computer industry moved toward multicore processors. At the core of this issue, considering the power density if the leakage of the transistors is not kept in check there could be threat of thermal run away causing the execution of all the cores at maximum or acceptable speed unfeasible. Having said this the computer industry is now in the dark silicon age where not all the cores of a processor can be functional with the highest power at the same time. So, this end of Dennard scaling and diminishing benefits from transistor scaling has propelled an era of Domain Specific Architectures.

With inefficiencies of general-purpose processors there are two clear opportunities. First, existing techniques for building software make extensive use of high-level languages with dynamic typing and storage management. As high-level languages are typically executed and interpreted inefficiently. Rewriting the code from python in C using parallel loops, caching, SIMD parallelism increases the performance by 47 - fold. However, A more hardware centric approach is to design architectures tailored to a specific problem domain and offer significant performance (and efficiency) gains for that domain and hence the name Domain specific architectures. This class of processors tailored for a specific domain—programmable and often Turing-complete but tailored to a specific class of applications are different from ASICs that are often used for a single function. Also called hardware accelerators, this achieve better performance and efficiency by exploiting parallelism for specific domain, effective use of memory hierarchy in contrast to less efficient caches, using dynamic precisions as and when needed and finally they benefit from targeting programs written in domain specific languages.

As such, accelerators are put in the spotlight to enable performance improvements necessary for emerging workloads. Although, most recently, accelerators have made their way into consumer electronics, edge devices, and cellphones (e.g., Edge TPU, NVIDIA Jetson, and Apple Bionic Engine), their limited computational capacity still necessitates offloading most of the inference tasks to the cloud. In fact, INFerence-as-a-Service (INFaaS)[18], has become the backbone of the deployed applications in Voice Assistants, Smart Speakers, and enterprise applications, etc. Cloud-backed inference currently dominates the market and is enabled by various forms of custom accelerators, such as Google TPU, NVIDIA T4, Microsoft Brainwave, and Facebook's DeepRecSys. As the demand for INFaaS scales one solution could be continuously increasing the number of accelerators in the cloud. Although intuitive, this approach is neither cost-effective nor scalable with the ever-increasing demand for DNN servic[18].

On the other hand, multi-tenancy, where a single node is shared across multiple requests, has been a primary enabler for the success of cloud-computing in current scale. In multi tenancy multiple instances of an application operate in a shared environment. The architecture works because each tenant is integrated physically, but logically separated. Without multi-tenancy, it is hard to even fathom the progress and future of datacenters and cloud-based computing. In fact, the broader research community invested more than a decade of efforts to develop solutions across the computing stack to bring forth seamless and scalable multi-tenant cloud execution models[11].

Nonetheless, multi-tenancy has not been a primary factor in the design of DNN accelerators because of the arms race to design the fastest accelerator, the utmost recency of accelerator adoption in datacenters, and challenges associated with multi-tenancy in accelerators. The datacenter accelerator designs revealed–for instance in Google's TPU or Microsoft Brainwave tend to show results focused on running a single neural network model as fast as possible[41]. Even the MLPerf benchmark suite keeps this single-model focus for both training and inference. But experience in cloud accelerator systems shows that keeping multiple models simultaneously resident on an accelerator has deployment benefits. Beyond just multiple customers sharing an accelerator, there is demand for multi-tenancy inside of a single application. For example, speech recognition and voice synthesis systems tend to require multiple models in deployment and can significantly benefit from multi-tenancy and co-location[11].

Google translate works using the principle of multi tenancy. It has been growing fast supporting few languages to 103, translating over 140 billion words a day. To make this possible, we needed to build and maintain many different systems in order to translate between any two languages. Google translate is now hosted on new system called google neural machine translation engine as shown in figure 4.1 which allows for Zero shot translation, where translation is done between language pairs which were never seen by the system explicitl[41].

Suppose we train a multilingual system with Japanese to English and Korean to English. The multilingual system, with the same size as a single GNMT system, shares its parameters to translate between these four different language pairs. This sharing enables the system to transfer the "translation knowledge" from one language pair to the others. This transfer learning and the

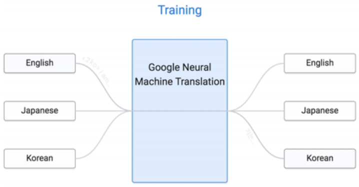


Figure 4.1. Google neural machine translation uses multitenancy

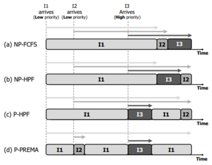


Figure 4.2. PREMA- time multiplexing DNN inferences

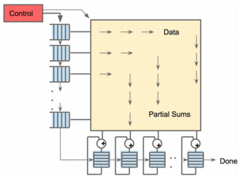


Figure 4.3. Systolic data flow of the Matrix Multiply Unit.

need to translate between multiple languages forces the system to better use its modeling power. In this case the GNMT works as a multi-tenant system sharing multiple models simultaneously on same node for improved performance. Yet, only this year has PREMA explored a scheduling algorithm that time-multiplexes a DNN accelerator across different DNNs through preemption as shown in figure 4.2. To amortize cost, cloud vendors providing DNN acceleration as a service to end-users employ consolidation and virtualization to share the underlying resources among multiple DNN service requests[42]. This paper makes a case for a “preemptible” neural processing unit (NPU) and a “predictive” multitask scheduler to meet the latency demands of high-priority inference while maintaining high throughput[38].

Preemptible NPU preempt the execution of a low priority task and allow high priority latency critical tasks. As shown in figure a preemptible NPU enables the higher priority I3 to finish earlier by proactively terminating low priority I1 task. Such preemption mechanism would enable intelligent scheduling policies that flexibly coordinate the allocation of shared resources among multiple inference tasks and meet target scheduling objectives.

Building on this, we have a predictive multitask scheduling algorithm(PREMA), that effectively balances latency, fairness, throughput and service level agreement. One main issue of preemptive NPU is in case we have a low priority task I2, can be starved from scheduling and experiences relatively more slowdown. If we were to predict the job length of I2, a better scheduling decision would be made for I2 to preempt I1, quickly finish its execution and allow I1 to resume its execution. This allows the average latency all tasks experience to be minimized while allowing the high-priority I3 to receive high-quality service. The predictable nature of computation, memory access details of DNN and NPU architecture allows us to develop a prediction model that estimates the job size of each inference task which is used to meet latency demands while not sacrificing throughput or SLA.

Planaria, on the other hand, sets to explore this timely, yet unexplored dimension of multi tenancy in the architecture design of DNN accelerators. This work presents a key idea of dynamically fissioning the DNN accelerator at runtime to spatially co locate multiple DNN interferences on the same hardware. Following are the main contributions. Dynamic architecture fission for spatial multi-tenant execution. Here we explore the dimension of dynamic fission in DNN accelerator[15]. This innovation enables simultaneous execution of multiple DNN acceleration threads to be spatially co-located on the same hardware substrate. This exclusive runtime reconfigurability in DNN acceleration offers a new degree of freedom in task scheduling to promote utilization and fairness while meeting the Quality of Service (QoS) constraints. Microarchitecture design for dynamic fission. The paper devises a concrete microarchitecture as an instance of dynamic fissionable architectures by delving into the design challenges associated with offering this technology on TPU like systolic design[38].

In this TPU systolic array design 4.3, as reading a large SRAM uses much more power than arithmetic, the matrix unit uses systolic execution to save energy by reducing reads and writes of the Unified Buffer. It relies on data from different directions arriving at cells in an array at regular intervals where they are combined. The above figure shows that data flows in from the left, and the weights are loaded from the top. A given 256-element multiply-accumulate operation moves through the matrix as a diagonal wave front. The weights are preloaded and take effect with the advancing wave alongside the first data of a new block[3]. Control and data are pipelined to give the illusion that the 256 inputs are read at once, and that they instantly update one location of each of 256 accumulators. From a correctness perspective, software is unaware of the systolic nature of the matrix unit, but for performance, it does worry about the latency of the unit.

Adding on the micro architecture design for dynamic design, we devise bi-directional systolic arrays for DNN acceleration that permits flow of data in all four directions from each element in the array. This low-cost additional flexibility expands the fission possibilities leading to significant energy reduction and performance gains. To coordinate fission with appropriate on-chip and off-chip data transfer, we arrange these bi-directional systolic arrays in on-chip pods that also comprise specialized interconnection and shared storage for each pod[10].

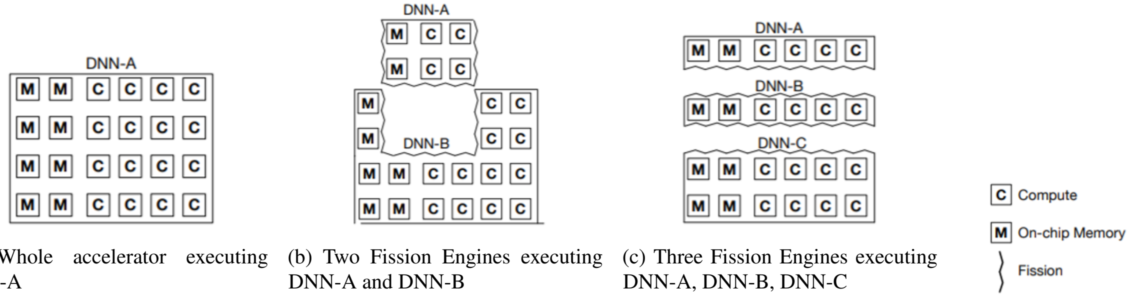
Task scheduling for spatial multi-tenant execution. To leverage architecture-level fission, the paper defines a task scheduling algorithm that breaks up the accelerator with respect to the current server load, DNN topology, and task priorities, all while considering the latency bounds of the tasks. As the following results indicate, this scheduling algorithm can harness fission capability to simultaneously co-locate DNNs to significantly improve utilization, throughput, QoS, and fairness[20].

We evaluate Planaria using three INFaaS workload scenarios made up of inference requests to nine diverse DNN benchmarks. Each scenario is evaluated under three different Quality of Service (QoS) requirements. We compare the proposed design to PREMA, a recent effort that offers multi-tenancy by time-multiplexing the DNN accelerator across multiple tasks. We use the same frequency, the same amount of compute and memory resource for both accelerators. Our results show that Planaria outperforms PREMA in terms of throughput by 7.4, 7.2, and 12.2 for soft, medium, and hard QoS constraints, respectively. For these set of constraints, Planaria also offers 45%, 15%, and 16% increase in Service-Level Agreement (SLA) satisfaction rate, respectively. At the same time, Planaria improves fairness by 2.1, 2.3, and 1.9. Our results suggest that exploring simultaneous spatial co-location through architecture fission and balanced task scheduling provides significant benefits. To this end, dynamic architecture fission paves the way for spatial multi-tenancy that can offer a unique direction in the era of cloud-scale acceleration of DNNs[8].

4.2 Dynamic Accelerator Fission

4.2.1 Concepts and overview

The objective is to enable multi-tenant execution of DNNs by spatially co-locating multiple DNN tasks on a single accelerator. To do so, the underlying accelerator needs to dynamically fission at runtime into smaller pieces of logical full-fledged accelerators that can execute their pertinent DNN. As shown in the figure there are three possible examples for the proposed accelerator fission and how the accelerator can spatially execute multiple DNN tasks simultaneously. Generally, a DNN accelerator is a collection of[37]



(a) Whole accelerator executing DNN-A (b) Two Fission Engines executing DNN-A and DNN-B (c) Three Fission Engines executing DNN-A, DNN-B, DNN-C

Figure 4.4. Illustration of possible fission schemes of Planaria with their corresponding spatially mapped DNNs

On chip memory banks and compute resource sources which is mainly a multiply and accumulate unit. The first figure 4.4a illustrates that if a DNN task with high priority or tight slack to meet the QoS constraints is dispatched to the accelerator, an entire accelerator is dedicated to the task to expedite its completion. In contrast 4.4b, 4.4c shows images that multiple DNN tasks being dispatched simultaneously. B has two fission engines and c has three fission engines, and there could be many other configurations in which fission into 2,3 components is possible. To process them all, the accelerator can fission into multiple logical accelerators, each of which executes a given task as shown. Importantly, fission needs to take place at both compute and memory level, since each logical engine is a standalone independent DNN accelerator. Moreover, the amount of compute and memory resources assigned to each logical accelerator ought to be balanced with the computational demand of the dispatched DNNs to maximize the throughput of the accelerator while meeting the QoS constraints. To that end, bringing forth spatial multi-tenant execution requires devising two major components the fission microarchitecture and task scheduler[34].

4.2.2 Fission microarchitecture

The first component of this work is a microarchitecture that can fission dynamically into smaller full-fledged accelerators to execute multiple DNNs simultaneously. A baseline monolithic DNN accelerator based on systolic array architecture and a set of challenges as well as the design requirements that should be considered to fission a monolithic design both at compute and on-chip memory level are discussed. Then we delve into the microarchitectural innards of Planaria, an incarnation of dynamic architecture fission. First, the design of Planaria adds bi-directional data movement in systolic arrays to offer variegated logical fission possibilities. Second, it uses this capability and a unique organization called Fission Pods to enable fission in systolic array based DNN accelerators[36]. Fission Pods are designed to offer a significant degree of fission flexibility, through specialized connectivity, on-chip memory organization, and bi-directional flow of data in its systolic units. This degree of flexibility is necessary to cope with the varying needs of dispatched DNNs that can be best matched by forming heterogeneous logical accelerators.

4.2.3 Task scheduler

As the second component of this work, we devise a task scheduling algorithm that adaptively schedules and assigns the resources to different tasks[7]. First, the scheduler identifies minimal amount of resources required to execute the DNN while meeting the QoS constraints imposed. Then, it uses a scoring mechanism that congregates task priority and remaining time to distribute the remaining resources on the accelerator to spatially co-locate tasks. This scoring mechanism leads to higher fairness as it considers multiple criteria and flexibility in the accelerator to co-locate multiple DNNs. Importantly, while the spatial co-location improves fairness, the scheduler effectively utilizes the dynamic fission mechanism and considers improving the QoS as its primary design principle. In fact, spatial co-location leads to better utilization of the accelerator resources as more than one task can run at the same time, which is in stark contrast with temporal approaches that rely on preemption.

4.2.4 Architecture Design for Fission: challenges and opportunities

The main design module is like a monolithic systolic accelerator like Google's TPU. Then we discuss a series a design decisions and requirements to enable spatial multi-tenant execution for DNNs.

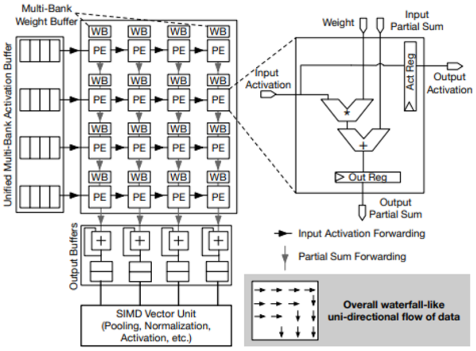

Multi-Bank Weight Buffer
Unified Multi-Bank Activation Buffer
WB
PE
Output Buffers
SIMD Vector Unit (Pooling, Normalization, Activation, etc.)
Weight
Input Partial Sum
Input Activation
Act Reg
Output Activation
Out Reg
Output Partial Sum
Input Activation Forwarding
Partial Sum Forwarding
Overall waterfall-like uni-directional flow of data

Figure 4.5. A monolithic systolic-array accelerator.

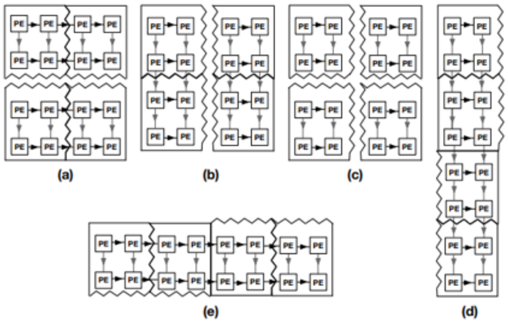


Figure 4.6. Illustration of possible fission scenarios.

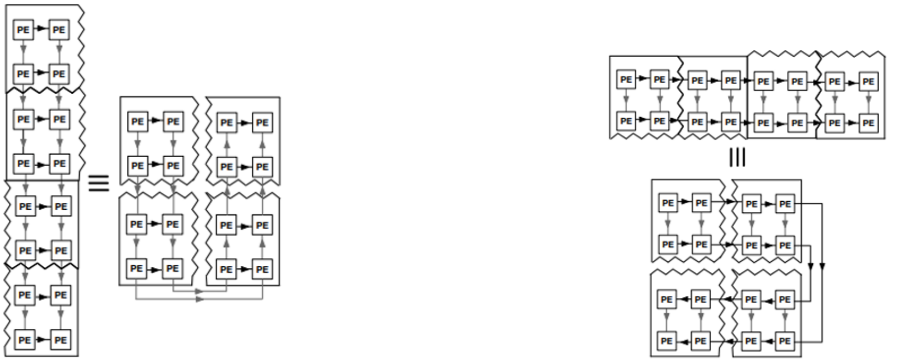
This figure 4.5 illustrates a model of monolithic systolic DNN accelerator. The accelerator consists of a 2D array of Processing Elements (PE) to perform matrix multiplications and convolutions, a unified multi-bank Activation Buffer, a 1D array of Output Buffers, and a SIMD Vector Unit to execute the remaining layers such as pooling, activation, batch normalization, and etc. Input activations are stored on-chip in the unified Activation Buffer–generally implemented as a multi-bank scratchpad, where each bank is shared across PEs within a row. Consequently, at each cycle, an input activation is read from an Activation Buffer's Bank and is reused for all the PEs within the row, where each PE is merely a Multiply and Accumulate (MAC) unit At each cycle, each PE forwards the input activation to the PE to its right (horizontal) and the output partial sum to the PE to its bottom (vertical). In short, this is a waterfall-like uni-directional flow of data as illustrated in the figure. Finally, the outputs are fed to the SIMD Vector Unit to perform further operations that are required by DNNs such as pooling, activation, batch normalization, etc. The remainder of the section elaborates on how to fission all the components comprising this monolithic accelerator. The objective is to fission the accelerator in such a way that strikes a balance between the performance of individual tenants while maximizing opportunities for other simultaneous tenants[34].

4.2.5 Fission for compute

There is a need for flexible fission in systolic to accommodate different algorithms execution on same substrate. Computational characteristics of DNNs such as data reuse and coarse-grained parallelism vary across different networks or even across different layers of a network. The systolic array architectures inherently exploit spatial data reuse for input activations along its rows and partial sums along its columns. However, a monolithic array design provides only a fixed dimension of this spatial data reuse. Moreover, as shown for TPU, mapping a convolution or matrix multiplication operation to a big monolithic systolic array can lead to underutilization of compute resources. As such, some layers naturally perform better if they are tiled to smaller chunks and parallelized across multiple smaller arrays, as that would exploit coarse-grain parallelism and yield better resource utilization. To that end, we first discuss how different fission configurations adapts to the characteristics of different layers through examples[39].

Figure 4.6 illustrates possible configurations for decomposition of a 44 systolic array, where a 22 subarray is used as the granularity for fission. Figure 4.6(a) shows a fission where the systolic array is broken down horizontally to two subarrays, while Figure 4.6(b) shows an instance of its vertical fission to two subarrays. Figure 4.6(c) illustrates another fission in both vertical and horizontal directions, yielding four systolic subarrays. For a layer that requires high coarse grain parallelism, fission in Figure 4.6(c) would be a good match, while Figure 4.6(b) would yield the best performance for layers that enjoy more partial sum reuse as well as the coarse-grain parallelism. In another scenario, if a layer requires high input activation reuse, moderate partial sum reuse, and coarse-grain parallelism, fissioning to Figure 4.6(a) will be the best choice[35].

Another important design decision is fission granularity where to break the systolic array . As illustrated in Figure 4.5, PEs are connected via two uni-directional links: horizontal and vertical. One extreme option is to replace these links to those that can be dynamically switched on and off to fission the systolic array at the granularity of a single PE. However, such a fine granularity of fission will impose significant overheads. Therefore, we instead replace a subset of the links to determine the granularity such that they can disconnect a subarray of the PEs instead of isolating a single PE. We need to perform a design space exploration and determines this fission granularity for the concrete microarchitecture[33].



(a) Down and up for partial sums

(b) Right and left for input activations

Figure 4.7. Bi-directional systolic execution.

Bidirectional systolic execution can be done for richer fission possibilities. As illustrated above 4.6 d and e shows two more fission scenarios[27]. If a network layer provides significantly higher opportunity in partial sum reuse than input activation reuse, while not requiring high parallelism, a scheme such as Figure 4.6(d) is desirable, while fissioning to Figure 4.6(e) will be a better design for significantly high input activation reuse. Realizing the last two configurations, however, requires additional design considerations. To forward the partial sums along four subarray fragments in Figure 4.6(d) and input activations in Figure 4.6(e), the data needs to flow in both directions: down and up for partial sums and right and left for input activations. Figure

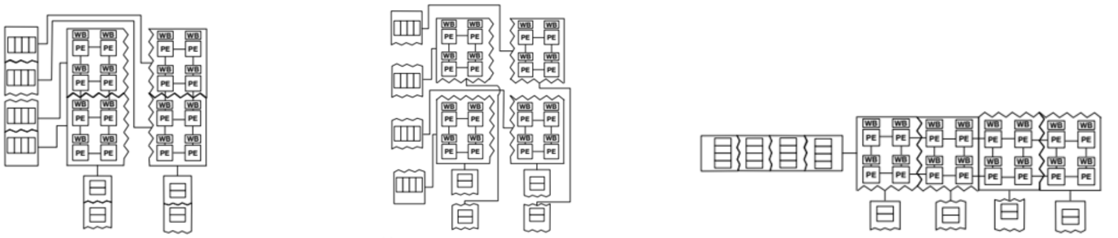
4.7a and Figure 4.7b illustrates how the partial sums and input activations need to flow at both directions to realize the desired scenario. To that end, we propose using bi-directional systolic arrays that can forward the input activations and partial sums in both directions as opposed to conventional systolic arrays that always forward the data in just one direction.[23] . In addition to the bi-directional intra-subarray data movement, we need inter-subarray communication to facilitate reconfigurability. As such, we exploit a bi-directional ring bus to connect the fission systolic subarrays. The links will be configurable in that they can be either off to fission two subarrays or on to forward input activations and partial sums.

Since the objective is to create stand-alone accelerators through dynamic fission, the SIMD Vector Unit also needs to be broken into smaller segments and coupled with each systolic subarray. Due to the parallel nature of this unit, we divide the original SIMD Vector Unit to smaller segments proportional to the number of systolic subarrays and designate a segment to each. When systolic subarrays need are vertically stacked (e.g., Figure 4.6(b,d)), a subset of these SIMD segments are bypassed[23].

4.2.6 Fission for on chip memory

In addition to the systolic array itself, the accelerator also requires fissioning the on-chip memory blocks to allocate commensurate storage to the compute units. Memory disaggregation across the chip is crucial for maximizing on-chip resource utilization [22]. While decomposing Weight Buffers is straightforward due to its coupling within the PEs, fission for the Activation Buffer and Output Buffer is more challenging. We need to divide the weight buffers as well. In systolic arrays, each PE harbors a private Weight Buffer that holds a subset of the network parameters. As such, the total Weight Buffer gets broken down naturally during fission as our strategy does not break the PE.

Figure 4.8 illustrates on-chip memory fission for three of the scenarios shown in Figure 4.6(b,c,e). Each of the scenarios requires different fission scheme for the Activation Buffer and Output Buffer as well as various patterns of connection between the buffers with the systolic



(a) Vertical fission

(b) Both vertical and horizontal fission

(c) Horizontal fission

Figure 4.8. On-chip memory fission and connection to subarrays.

subarray, which are not possible in a monolithic design[16]. In the monolithic case, the Activation Buffer is just connected to the leftmost PEs and Output Buffer to the bottom-most PEs. However, as Figure 4.8 depicts, more patterns of connectivity between these buffers and the PEs/subarrays are necessary. To support these variegated patterns, we devise a microarchitectural block, dubbed Fission Pod, where the Activation Buffer and Output Buffer are co-located in a memory substrate that is shared amongst a group of connected bi-directional systolic subarrays[5].

4.3 Microarchitecture for Fussion

This section delves into the microarchitectural design that enables the dynamic architecture fission for spatial multi-tenant execution. We first discuss the design for the proposed bi-directional systolic array and Fission Pod, then highlight the overall architecture of the proposed accelerator, Planaria, that constitutes multiple Fission Pods.

4.3.1 Bidirectional systolic array design

Bi-directional data movement for both input activations and partial sums in the systolic array provides additional decomposition and rearrangement possibilities that can lead to better utilization. To enable this additional movement, each PE in the systolic array should be able to also send input activations to the PE to its left and partial sums to the PE located above. Figure 4.9 illustrates how a set of of additional multiplexers and de-multiplexers around a PE can enable bi-directional movement[14]. As highlighted in black, a multiplexer at the left of the PE selects the inputs from either the activation coming from the right or the left. A de-multiplexer at the right selects which of the left or the right PE should receive the activation. The multiplexer and the de-multiplexer are coupled and are controlled by the same single bit, that sets the direction of the input activations in the systolic array. Similarly, another pair of multiplexer/de-multiplexer on the north and south of the PE in the Figure 4.9 control the flow of partial sums. To enable fission and bi-directional inter-subarray data movements[2], PEs at the boundaries of systolic subarrays are also connected through these multiplexer/de-multiplexer pairs to the corresponding PEs in the adjacent subarrays. Synthesis results show that these extra logics are not on the critical path that determines the clock cycle and are local addition that do not timing issues. The critical path is from the Weight Buffer to the Output Register, where access to the on-chip buffer dominates the execution time in a single cycle.

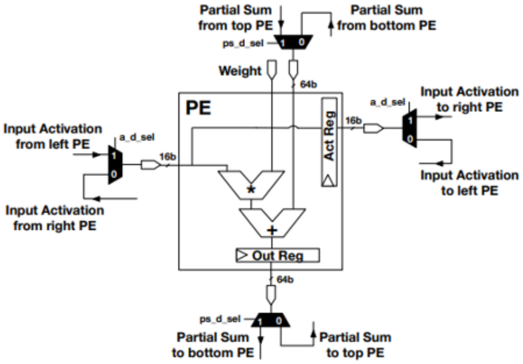


Figure 4.9. Switching network for bi-directional systolic array

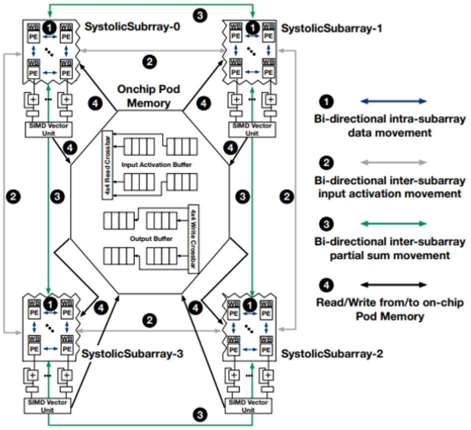


Figure 4.10. Overall architecture of Planaria

4.3.2 Fission Pod design:

To address the challenges discussed in Section 3.2 for fission of memory along with the systolic array, we propose a microarchitectural unit, called Fission Pod, which interweaves the on-chip memory with the systolic subarrays and provides balanced cooperation of these components. Figure 4.10 illustrates the design. As shown, at the center of this unit, an on-chip memory substrate, called Pod Memory, is placed, and connected to a group of systolic subarrays.

Following discusses the cooperation and communication of the subarrays and the Pod Memory. A conventional systolic array harbors a unified multi-bank Activation Buffer and a unified multi-bank Output Buffer on their left and bottom, respectively (see Figure 4.5). When a systolic array is broken to four subarrays as depicted in Figure 4.10, the buffers are moved to Pod Memory and are broken down to four corresponding independent multi-bank buffers[40]. These four buffers are connected to the four systolic subarrays via two 44 crossbars to maximize flexibility and fission possibilities that require various patterns of connectivity between on-chip buffer and systolic subarrays. One crossbar is for reading from Activation Buffers and the other for writing to Output Buffers of the Pod Memory. The choice of crossbar here is to facilitate the diverse communication patterns, some which were illustrated in Figure 4.8. As an example, for fission configuration in Figure 4.8(b), when the subarrays operate independently, the crossbar connects each of them exclusively to one of the four Activation Buffers to read the input activations and to one of the four Output Buffers to write the outputs. Alternatively, for the fission scheme in Figure 4.8(c), when we reconstruct a wide systolic array, the four Activation Buffers need to act as a unified memory and are used in a round robin fashion, effectively utilizing the entire Activation Buffers in the Pod Memory[19].

Besides the memory-compute communication within the Fission Pod, the systolic subarrays are also connected to one another via two sets of bi-directional ring buses. One bus is to pass activations between bi-directional subarrays(2) and the other is to forward the subarray partial sums(3). These buses enable realizing different fission possibilities while leveraging the bi-directional nature of our subarrays. For instance, to realize the fission scheme in Figure 4.6(e), where the subarrays reconstruct a wide array, the activation ring bus will chain the subarrays. The SystolicSubarray-0 in Figure 4.10 sends the activations to SystolicSubarray-1, and so on and so forth. Since for fission scheme in Figure 4.6(e), there is no need for partial sum forwarding, the partial sum ring bus will be switched off. The two ring buses are pipelined to alleviate any potential critical paths due to the connectivity between the subarrays[21].

4.4 Planaria Overall Architecture

Figure 4.11 illustrates the overall architecture of our proposed accelerator, Planaria. As shown, the original monolithic systolic array has been broken down to 16 bi-directional systolic subarrays, where a group of four subarrays form one Fission Pod that contains a Pod Memory. All these 16 subarrays are connected globally along the accelerator chip via the afore mentioned bi-directional ring busses for input activations and partial sums data movement. Hence, in one extreme, all these ring busses can be switched on to construct the biggest logical accelerator, running only one DNN on the entire accelerator. Alternatively, in another extreme, all the ring buses can be switched off to provide 16 standalone logical accelerators, spatially co-locating 16 different DNNs simultaneously for multi-tenant execution. Overall, this architecture supports 65 fission scenarios that can simultaneously co-locate various number of DNNs from 1 to 16. Each of the four Fission Pods is connected to one off-chip memory channel. The bus that brings the data from off-chip memory channel goes around the subarrays and can fill their weight buffers. This bus is also connected to the Pod Memory to load/store intermediate activations/output to/from the off-chip memory channel. This bus is pipelined and is no different than the bus that feeds the off-chip data to a systolic array. To avoid clutter, Figure 8 does not illustrate the off-chip memory buses[1]. The Fission Pods are connected to their neighbors through a direct link that can foster data reuse to reduce costly off-chip accesses. If data is present in one of the pods, it can be sent to another at most with two hops. Planaria can fission up to 16 logical accelerators and therefore, it can simultaneously co-locate 16 different DNNs. However, depending on the combination of the co-located DNNs, 65 total fission scenarios are possible. A logical accelerator, which represents one of these 65 possibilities, can encompass multiple physical Fission Pods. A logical accelerator can either work as a logical monolithic systolic array or further fission if a DNN layer benefits from coarse-grain parallelism. Planaria's interconnections and bus are designed such that, a logical accelerator can take a portion of a Fission Pod and another logical accelerator takes the rest. In Figure 8, one logical accelerator that accelerates DNNA can comprise the subarrays in Fission Pod-0 with two subarrays from Fission Pod-3 (Fission Pod-3.SystolicSubarray-0 and Fission Pod-3.SystolicSubarray-1). The remaining two subarrays from Fission Pod-3 can form another logical accelerator to accelerate DNNB.

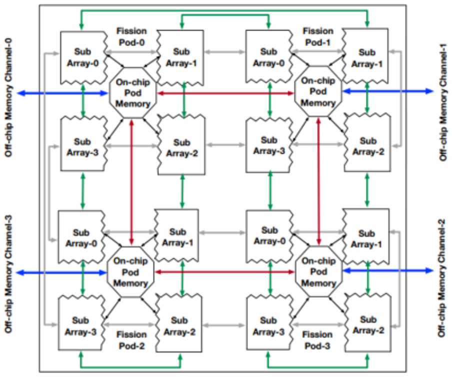


Figure 4.11. Overall architecture of Planaria.

Dynamic reconfiguration for fission and multi-tenant execution: Conventional systolic arrays operates in tile granularity. That is, they fetch a tile of weights and activations and produce a tile of intermediate activations or outputs. Planaria does not deviate from this convention. Consider a scenario where three DNNs are simultaneously co-located with some fission scheme on Planaria, and fourth DNN is now dispatched to be accelerated. In this case, Planaria allows the old three co-located DNNs finish computing the tile that they are processing. In the meantime, the scheduler decides the new allocation of the subarrays considering the newly dispatched DNN. At the same time, Planaria loads this new fission configuration as a set of bits that decides the direction of the subarrays and the off/on connectivity state of the buses. Each Planaria subarray requires two 6-bit registers, one retaining the current configuration state and the other pre-holding the next state. Six bits is sufficient for reconfiguration of each subarray and its directions/buses. One bit determines the direction of input activation, another sets the direction of partial sums. Each subarray can potentially connect to four other subarrays, which can be in the neighboring Fission Pods. Four bits determines which ones are going to be connected. The direction of connectivity can be deduced from the direction of the subarray. Another eight bits determine the connectivity of the Pod Memory buffers to the subarrays in the same Fission

Pod. Like conventional systolic design, each subarray is equipped with an instruction buffer and a Program Counter (PC) that indicates the current macro instruction. While the subarray is draining the instructions for the old DNNs, Planaria fetches the next instructions associated with the new configuration. Starting the execution with the new fission configuration becomes simply adjusting the PC to the first instruction associated with the new configuration. The mechanism is no different than prefetching the instructions for a new tile in conventional systolic arrays. The difference is that, each subarray has a designated PC and a designated 4 KB instruction buffer. For INFaaS, since each DNN will serve unbounded set of inference requests, it is intuitive to precompile the DNN and run the precompiled binary again and again. For each layer in a DNN, Planaria can allocate between 1 through 16 subarrays. Although allocation of a few subarrays provides more fission possibilities, only one is optimal for the layer given a certain number of subarrays. Therefore, to facilitate scheduling, the DNNs can be precompiled to these 16 different logical accelerators resulting from fission.

4.5 Spatial task scheduling

The primary goal of the Planaria's task scheduler is to fully leverage the dynamic fission capability to maximize performance in terms of utilization, throughput, while meeting the QoS constraints. This section delineates the overall flow of the proposed spatial task scheduling algorithm, in Algorithm 1. Overall flow. To leverage the dynamic fission provided by the architecture, the scheduler is invoked whenever (1) a new inference task is dispatched to the task queue of the datacenter node or (2) a running inference task finishes. As mentioned in the previous section, the scheduling events happen at tile granularity. Each scheduling events consists of the following two major stages. Given the DNNs in the task queue, the first stage determines the minimum amount of resource (number of subarrays) necessary to meet the QoS requirement for tasks. Given this information, the second stage determines the allocation of the subarrays based on their availability and priority of the inference requests. This high-level flow of the scheduling is shown in function SCHEDULETASKSPATIALLY of Algorithm 1.

We need to estimate minimal resource to meet the QoS requirement. This algorithm exploits the dynamic architecture fission by adaptively assigning resources about the intrinsic slack times provided by each DNN inference task. The algorithm begins by first identifying the minimal resources required to meet the QoS requirement. As we know the list of possible configurations and the slack time of each task a priori, we feed these information to a prediction model similar to [4, 37, 56] to predict the execution time for different logical accelerator configurations and use the outputs to deduce the minimal resource to meet the QoS requirement. Performance estimation is viable since the topology of the DNN does not change, there is no hardware managed cache, or dynamic control flow in the execution. The EstimateResources function in Algorithm 1 summarizes this stage. Also, resources need to be allocated to improve QoS. After identifying minimal resource for each task, we determine whether all the tasks in the queue can be co-located simultaneously. Depending on whether all the tasks can be spatially co-located on Planaria, this stage invokes two different functions, ALLOCATEFITTASKS and ALLOCATEUNFITTASKS , as shown in line 6–10 in Algorithm 1. First, when all the tasks can be spatially co-located, the function ALLOCATEFITTASKS will first assign the minimum number of subarrays required to meet the QoS requirements. Then, if there are remaining resources, the scheduler aims to optimally distribute these spare resources using a score function that balances priority and the remaining time of each task, as shown in line 27 in Algorithm 1. Consequently, this score function not only fosters throughput but also the fairness among the tasks. Finally, the scheduler allocates the spare resources proportional to the score of each task. On the other hand, when only subset of the tasks fit on Planaria, the scheduler uses the ALLOCATEUNFITTASKS function to resolve the competition among the tasks. Like the approach used to assign the spare resources, the function leverages a score that uses priority, slack, and the minimum required resource of each task, as shown in line 40 in Algorithm 1. This scoring mechanism gives advantages to the tasks with higher priority to improve fairness, and to the ones with less slack time or less resource requirement to maximize QoS satisfaction and throughput. Finally, the scheduler, allocates the

Algorithm 1. Spatial Scheduling for Planaria

```
function ScheduleTasksSpatially($\Omega$)
    $estimate \leftarrow \{\}$
    for $task$ in $\Omega$ do
        $estimate[task] \leftarrow$ EstimateResources($task$)
    end for
    if $Planaria.fits(estimates)$ then
        $s \leftarrow$ AllocateFitTasks($\Omega, estimates$)
    else
        $s \leftarrow$ AllocateUnfitTasks($\Omega, estimates$)
    end if
    return $s$
end function
```

```
function EstimateResources($task$)
    $candidate \leftarrow []$
    $slack = task.QoS\_constraint - task.executed\_time$
    for $num\_subarray$ in $range(Planaria.size)$ do
        if PredictTime($task$) $\leq slack$ then
            $candidate.append(num\_subarray)$
        end if
    end for
    return $min(candidate)$
end function
```

```
function AllocateFitTasks($\Omega, estimates$)
    $allocation \leftarrow \{\}, scores \leftarrow \{\}$
    for $task$ in $\Omega$ do
        $allocation[task] \leftarrow estimates[task]$
        $score[task] \leftarrow \frac{task.priority}{task.remaining\_array}$
    end for
    $remaining\_array \leftarrow Planaria.size - \sum estimates$
    for $task$ in $\Omega$ do
        $fraction \leftarrow \frac{score[task]}{\sum score}$
        $allocation[task] += fraction \times remaining\_array$
    end for
    return $allocation$
end function
```

```
function AllocateUnfitTasks($\Omega, estimates$)
    $allocation \leftarrow \{\}, scores \leftarrow \{\}$
    for $task$ in $\Omega$ do
        $slack \leftarrow task.QoS\_constraint - task.executed\_time$
        $score[task] \leftarrow \frac{task.priority}{slack \times estimates[task]}$
    end for
    $scores.sort(reversed = True)$
    $remaining\_array \leftarrow Planaria.size$
    while $remaining\_array > 0$ do
        $allocation[task] \leftarrow estimate[task]$
        $remaining\_array -= estimates[task]$
    end while
    return $allocation$
end function
```

resources to different tasks in the order of their scores until Planaria becomes fully occupied.

4.6 Evaluation

4.6.1 Methodology

Benchmark DNNs. Following the methodology presented in MLPerf, a recent effort in benchmarking deep learning systems and applications, we choose our representative DNN models, from three domains of image classification, object detection, and machine translation. We use nine diverse DNN models from the aforementioned domains to construct a set of DNN tasks with various layer dimensions and types of operations (regular and separable depth-wise/pointwise convolution, LSTM, batch normalization, fully connected, pooling, residual, etc.), including recent and state of-the-art deep neural models such as Efficient Net and YOLOv3. Multi-tenant workloads. Commensurate with MLPerf, as Table 4.6 shows, we create three INFaaS workload scenarios made up of inference requests to the benchmark DNNs: (a) Workload-A (from requests to ResNet-50, GoogLeNet, YOLOv3, SSD-R, and GNMT); (b) Workload-B (from requests to EfficientNet-B0, MobileNet, SSDM, and Tiny YOLO; and (c) mixed weight Workload-C (from request to all the nine DNNs). To generate multi-tenant instances from these workload scenarios, we assign a random arrival time for each inference request by drawing samples from a Poisson distribution, commensurate with the methodology of MLPerf and other works to mimic task dispatching in datacenter servers. We assign priority levels to the dispatched tasks by drawing samples from a uniform distribution. The priority levels are within the range of 1 to 11. We use Quality of Service (QoS) constraints presented by MLPerf for the server scenarios. To well exercises our proposed system, we use three levels of QoS for each workload scenario, (a) QoS-S as a soft QoS constraint (defined as 1 QoS given in MLPerf), (b) QoS-M as a medium constraint (1 4× QoS), and (c) QoS-H as a hard constraint $(1/16 \times QoS)$ to evaluate sensitivity with regard to QoS latency constraints.

Hardware modeling. We implement the proposed bidirectional systolic subarray and the

Table 4.1. Workload scenarios and benchmark DNNs from three domains: image classification, object detection, and machine translation.

Workload	Load Weight	Domain	DNN Model(Release year)
Workload Scenario-A	Heavier	Image Classification	ResNet-50(2015),GoogLeNet (2014)
		Object Detection	YOLOv3(2018), SSD-R(2016)
		Machine Translation	GNMT(2016)
Workload Scenario-B	Lighter	Image Classification	EfficientNet-B0(2019), MobileNet-v1(2017)
		Object Detection	SSD-M(2017), Tiny YOLO (2017)
Workload Scenario-C	Mixed	Image Classification	ResNet-50, GoogLeNet, EfficientNet-B0, MobileNet-v1
		Object Detection	YOLOv3, SSD-ResNet34, SSD-MobileNet, Tiny YOLO
		Machine Translation	GNMT

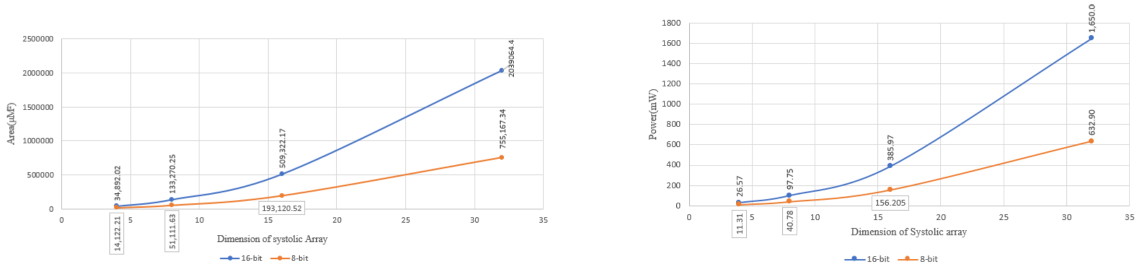
bussing systems including crossbars for the Fission Pods in Verilog and synthesized them with Synopsys Design Compiler using FreePDK-45nm standard cell library to extract their power/area. We model the on-chip SRAM using CACTI-P that provides energy and area. The on-chip busing system is modeled using McPAT 1.3 and the energy cost estimated to be 0.64 pJ/bit per hop. Simulation infrastructure for Planaria. We compile each DNN benchmark to Planaria and develop a cycle-accurate simulator that provides the cycle counts and statistics for energy measurements for each DNN using the modeling described above. We include all the overheads of reconfiguration, fission, instruction fetch, off-chip memory accesses, etc. We verify the cycle counts with our Verilog implementations. Comparison with PREMA. We compare our proposed Planaria accelerator that supports spatial multi-tenant execution of DNNs to PREMA that exploits preemption mechanisms to support multi-tenancy via temporal execution. Baseline PREMA utilizes a monolithic TPU-like systolic array DNN accelerator as its hardware. For the sake of fairness in comparison, we use the same number of PEs (128128=16,384), on-chip activation /weight/output buffers (12 MB), frequency (700 MHz), and off-chip memory bandwidth as reported in PREMA's work for our design. The detailed analysis of the synthesis results shows that our design can meet 1GHz frequency and the added bi-directional links or the buses, due to pipelining, are not on the critical path. However, for fair comparison with PREMA, we still use their reported 700 MHz frequency, which is based on TPU. PREMA's monolithic systolic array is not explicitly optimized to execute the most recent DNNs that use depth wise convolutions such as Efficient Net and Mobile Net. A recent study on designing systolic array generators form UC Berkeley also makes this observation and suggests executing these specific layers on CPUs instead of systolic arrays. Unfortunately, our experiments show CPU execution to be slightly slower (4%) than the monolithic systolic array we evaluated. In our evaluation, Workload-A does not include any DNNs with separable depth-wise convolutions. Evaluation Metrics. To evaluate the effectiveness of the proposed solutions, we use the following metrics: Throughput is defined as the maximum queries-per second (1λ) achieved by the system according to Poisson distribution (λ) while meeting the SLA for different QoS constraints (QoS-S, QoS-M, and QoS-H). According to MLPerf, meeting SLA is defined as executing an image classification or object detection task 99% of the time and a translation task (e.g. GNMT) 97% of time within its QoS latency bound in a multi-tenant workload. This is the main metric for evaluation of server scenarios for inference tasks in MLPerf. SLA Satisfaction Rate is the fraction of multi-tenant workloads that adhere to the SLA described above. Fairness measures the equal progress of the tasks in a multi-tenant setting while considering task priorities. We use the same definition for fairness given in PREMA baseline, as

$$fairness = min_{i,j}\frac{PP_i}{PP_j}, \text{while } PP_i = \frac{T_i^{isolated}}{T_i^{multi-tenant}} / \frac{Priority_i}{\sum Priority_k} \tag{4.1}$$

Energy reduction compares total energy consumption to run multi-tenant workloads on both Planaria and PREMA.

4.7 Design Space Exploration

Various analysis was done for systolic arrays to understand the power and area overhead. Dimension tested for unidirectional array to understand the design space. Sizes tested were 4,8,16,32. And bit widths tested are 8, 16. Analysis was done to find the relation between dimension area and power for interpolating between various sizes of fissioned architectures. From figure 4.12 we can see doubling the dimension quadruples the SA area. Bit-width also relies on forms a linear relation, by doubling bitwidth the area doubles. So, we can see there is a scope for optimizing the area and power by dynamically modifying the bit-widths. This could be part of future work. Power follows similar trends and does not account for any significant overheads or disparity. A size 128X 128 number of PEs were considered. The size of subarrays was swept from 16x16, 32x32, 64x64. And to find the optimal size the energy delay product was considered. The relative EDP values for three design points was considered and 32x32 offers least EDP. So, we use this size for fission granularity. From 32 as we increase the size, we will not be able to exploit sufficient parallelism. However, as we reduce the size, the benefits of parallelism are outweighed by the additional switching circuitry.



(a) Unidirectional area trend

(b) Unidirectional power trend

Figure 4.12. Unidirectional parameter trends

Bi-directional data movement for both input activations and partial sums in the systolic array provides additional decomposition and rearrangement possibilities that can lead to better utilization. To enable this additional movement, each PE in the systolic array should be able to also send input activations to the PE to its left and partial sums to the PE located above. Figure 4.13 illustrates how a set of of additional multiplexers and de-multiplexers around a PE can enable bi-directional movement. As highlighted in black, a multiplexer at the left of the PE selects the inputs from either the activation coming from the right or the left. A de-multiplexer at the right selects which of the left or the right PE should receive the activation. The multiplexer and the de-multiplexer are coupled and are controlled by the same single bit, that sets the direction of the input activations in the systolic array. Similarly, another pair of multiplexer/de-multiplexer on the north and south of the PE in the Figure control the flow of partial sums. However, we have one additional set of multiplexer for partial sum, as we have to decide if we want to pass on the partial sums or accumulate them.

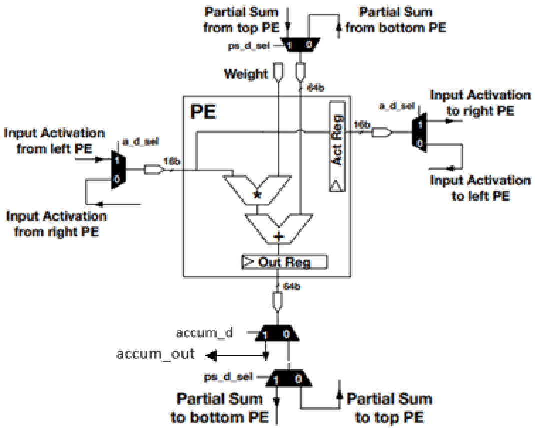


Figure 4.13. Bidirectional systolic array

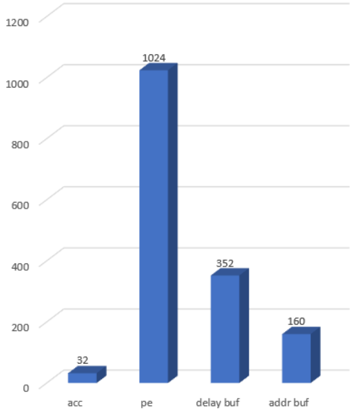
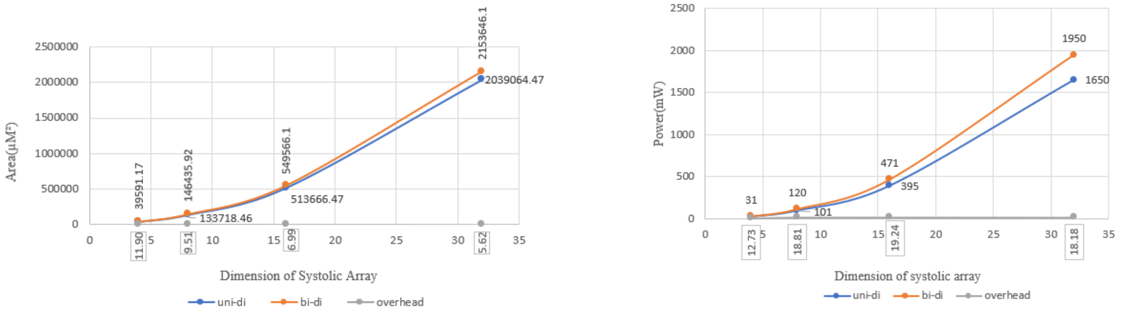


Figure 4.14. Quantity of each component

Design space exploration was done on bidirectional systolic arrays to understand the power and area overhead because of the switching circuitry as shown in figure 4.15. This overhead mainly comprises of the additional multiplexers and demultiplexers. Bidirectional SA are swept for value 4, 8,16,32- and for-bit widths 8, 16. We see around 7% area over head on an average owing for the control logic. This is a good tradeoff for the additional flexibility that we obtain for the fission architecture. However, in case of power we see around 18% increase. The power increase was slightly more than expected. However, we know power is proportional to area and needs to increase with similar trends. So, to understand this we had to split energy consumption to see if this was expected as shown in table 4.2. We analyzed three different parts of the power mainly switching, internal power. So, leakage power is same as static power which occurs whenever there is direct shot to ground. internal power is the dynamic power dissipated within the cell boundaries. Like charging and discharging of nodes. Switching power is hidden power which is the input switching power when outputs are not switching. Like when there is a clocking switching activity on the cell inputs and the output is not changing. We see that the internal power overhead is constant as expected with around 12% power overhead which is expected. But however, the switching power and leakage power overheads are not constant, and they are increasing monotonically. This could be the reason for such high-power overheads.



(a) Area trends

(b) Power trend

Figure 4.15. Bidirectional

Table 4.2. Table represents power split for all categories.

dimension	Bi directional SA			uni directional SA			overhead		
	switch	internal	leakage	switch	internal	leakage	switch	internal	leakage
4	8.16	21.7	0.79	8.02	18.7	0.76	1.746	13.825	3.7975
8	34.2	77	3.18	31.1	67.2	2.91	9.265	12.727	8.4906
16	142.3	314.4	4.7	123	271	4.1	15.69	13.804	12.766
32	670	1240	5.3	529	1080	4.42	26.65	12.903	16.604

We tried exploring one more aspect for our fission architecture. To check the overheads for connecting two SAs vs a monolithic SA as shown in table 4.3 and 4.4. Three configurations were tested. 32x64, this uses activation reuse. 64x32 this has partial sum reuse. 64x 64 which can have both reuse and no re use at all. 32x64 and 64x32 are similar architectures however, 64x32 the one with partial sum reuse has slightly more area overhead than its counterpart, this is because we have an additional multiplexer which checks to see if the output needs to be forwarded or accumulated in the SA. Now comparing the difference between monolithic, fissioned and connected SAs. The overheads have been calculated for three different configurations again. Here monolithic is the standalone 32X64 or 64x32 architecture. Fissioned are normal 32x32 architecture. There values have been computed by just taking their individual area and multiplying by the factors they have been replicated. And connected architecture areas are obtained by designing a circuitry where 2 32x32 systolic array are connected via pins and synthesizing the design. We see not much area over head between connected and monolithic showing that which performing fission we just need to account for the switching circuitry. But fission trends are not similar as we are just multiplying, and this does not account for the buffer sizing and optimization that are done when area increases. Performing the same analysis on power shows that there is a similar trend between both.

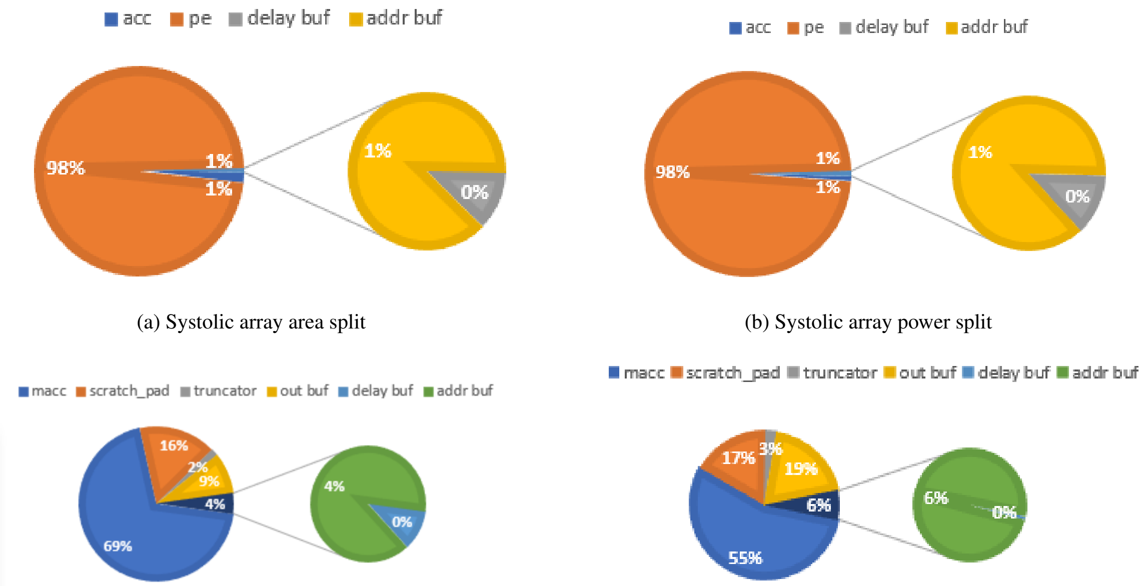
Table 4.3. Area overhead between Monolithic SA, connected SA (multiple instances connected in top module), fissioned SA (times single unit) measured to show the benefit of fission.

Area trend	**dim**	**Monolitic SA**	**connected SA**	**fissioned SA**	**1-3 overhead**	**2-3overhead**
action reuse	32-64	1682527.11	1682836.97	1689866	0.43618257	0.47689302
partial sum reuse	64-32	1688648.04	1688928.15	1689866	0.01658783	0.055529301
no reuse	64-64	3369216.87	3369829.32	3379732	0.01817781	0.293862954

Table 4.4. Power overhead between Monolithic SA, connected SA (multiple instances connected in top module), fissioned SA (times single unit) measured to show the benefit of fission.

power trend	**dim**	**Monolitic SA**	**connected SA**	**fissioned SA**	**1-3 overhead**	**2-3overhead**
action reuse	32-64	1620	1645	1664	2.71604938	1.155015198
partial sum reuse	64-32	1750	1781	1664	1.77142857	-6.56934307
no reuse	64-64	3400	3446	3328	1.35294118	-3.42426001

Further analysis was done to understand the power and area trends obtained as shown in figure 4.16. Most of the area inside the SA architecture is because of the PE, this can be seen from the quantity graph and this part is expected as it is the main compute units and cannot be avoided. The area and power graphs for PEs are in proportion. There are 32 accumulators one for each column. And they comprise around 1% of total area and power, similarly, buffers account for 1% of total area and we see similar results for power. Now dwelling deeper into PE architecture, we see that its mainly comprised of multiply and accumulate units and used for computation. It comprises 70% of the systolic array. And rest of the area is mainly due to the buffers. One observation that we see here is that power distribution of buffers is slightly more than the area distribution. This could be because of the hidden power where power if consumed for changes in input without change in output. These latches are driven by the clock



(a) Systolic array area split

(b) Systolic array power split

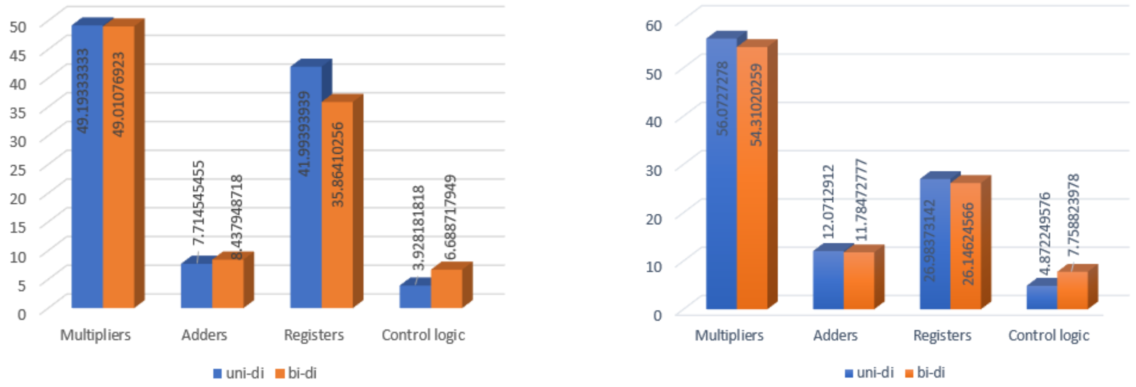
(c) Individual PE area split

(d) Individual PE power split

Figure 4.16. Systolic array, individual PE parameter trends

which keeps continuously changing even though the output might not change causes extra power consumption. Clock gating in this scenario would help achieve lot of power saving. Finally, higher level division was performed to see the overall distribution between compute, memory, and control logic as shown in figure 4.17. We do not see much change in uni directional and bidirectional between multipliers adders and buffers which is expected. However, control logic area increases by 60% and power by 70%. However, this additional circuity is the trade off we make for flexibility in the fission architecture.

The communication between pod memory and subarrays is crucial and needs to be very fast. We are using cross bar switches for this purpose as they support diverse communication with minimum time delay. Three designs as shown in figure 4.18 were considered to understand the area and power overheads as shown in figure 4.19. We measure this for a 4x4 which was selected as our fission granularity is 2 and we have 4 fission pods. Coming to the first design we have 4 inputs to multiplexer and connected to a demultiplexer with 4 outputs. However, there is only one channel between them. So, whenever there is just one input this is the best architecture as it consumes less area. However, if there are 4 inputs simultaneously then we have to time multiplex

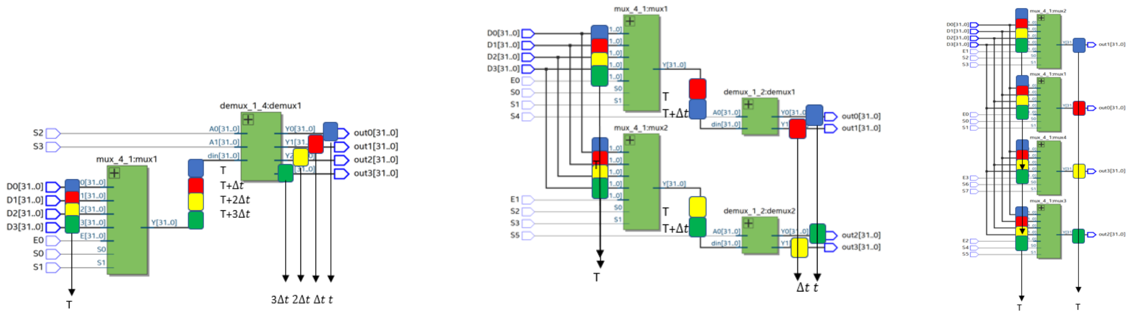


(a) High level power comparison

(b) High Level Area Comparison

Figure 4.17. High level unit distribution

between all the inputs and worst case it would take 4-time units for the input to transfer to output. In design 2 we have 4 inputs multiplexer connected to 2 separate 1:2 demultiplexers[17]. In this case we have 2 separate channels and the output in worst case is time multiplexed between 2 set of inputs. So worst case time delay is 2 units. The final design we have only 4 multiplexers and this consumes the largest area as there are 4 separate multiplexers. But however, in this case as soon as we get the input it is transferred to output, we do not have to wait for the inputs. And the scratchpad connected to the sub arrays they need to be as fast as possible we consider using this design. Cross bar area and power trends were measured for 2, 4, 8 just to check the variation with respect to size as shown in table 4.5. We see that they are linearly proportional.



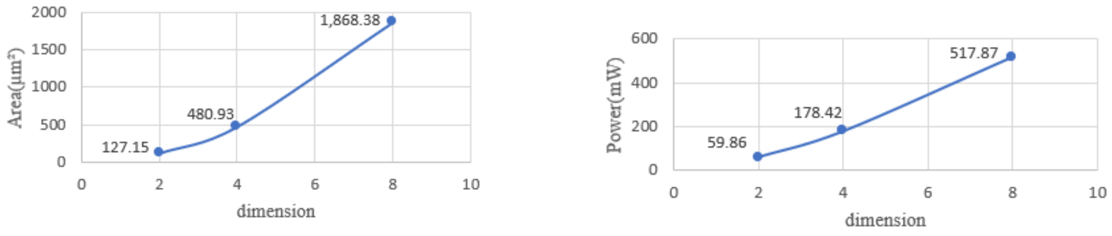
(a) Design1

(b) Design2

(c) design3

Figure 4.18. Crossbar designs.

Control logic is the crucial part in our fission architecture. We have multiple switches like



(a) crossbar area trend

(b) crossbar power trend

Figure 4.19. crossbar parameter trends

Table 4.5. crossbar sizing for inputs:2,4,8

config	area	power
crossbar_t	480.92	178.423
crossbar_2t	393.68	154.911
crossbar_4t	246.84	111.928

Table 4.6. Critical paths for memory configurations

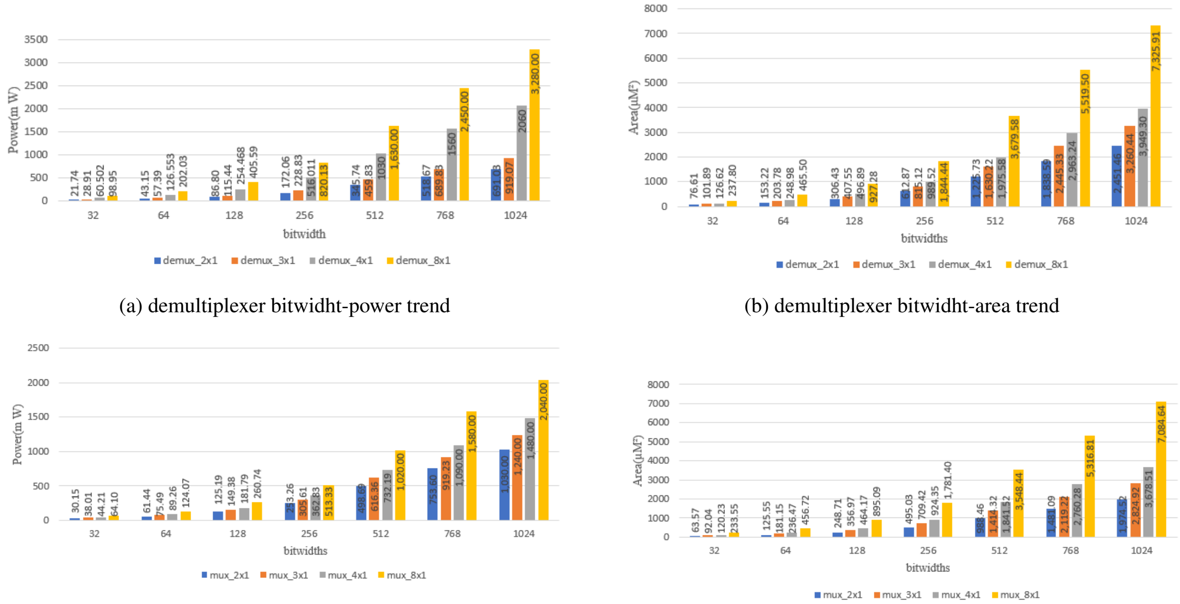
memory	Delay(CP)
scratchpad	4.087ns
Dual port SRAM	4.093ns

the crossbars for communicating between the pod memory and subarrays, there is also intra sub array, inter subarray activation and partial sum data movement. Read/write to on chip pod memory. All these needs to dynamically turned on and off dynamically fission the architecture as and when needed. and for bidirectional data flow we need extra control logic. So extensive design space exploration was done to find the switch configurations. 2:1,3:1,4:1, 8:1 multiplexer and demultiplexers where synthesized to find the best area power configuration in our design. Also, it is swept across various bitwidths like 32, 64, 128, 256, 1024, 2048. Following are few switch configurations and bitwidths as shown in table 4.7 are selected for the design. As discussed, design was swept for 2-1,3-14-18-1 multiplexer and demultiplexers for all the bit widths as shown in figure 4.20.

Table 4.7. Final switch configurations used

	bits	area(μM^2)	power(mW)
multiplexer_4X1	768	2760.282	1090
demultiplexer_1X4	768	2963.239	1560
crossbar_4X4	1024	14714.056	5940
crossbar_4X4_out	4096	58856.224	2376
multiplexer_3X1	768	2119.22	919.228
demultiplexer_1X3	768	2445.326	689.8271

Double buffering was used to make the weight buffers. It is mainly used in streaming



(a) demultiplexer bitwidht-power trend

(b) demultiplexer bitwidht-area trend

(c) Multiplexer bitwidht-power trend

(d) Multiplexer bitwidht-area trend

Figure 4.20. parametric sweep for different inputs and bit-widths

algorithms and for high performance computing. The idea is that instead of reading and then writing using a single port ram, we can use them in double buffering configurations to do simultaneous reads and writes. There by saving clock cycles. So here we have 2 single ports rams, at a point in time compiler will be writing to one SRAM and reading from the other SRAM. Once the reading is done completely exhausting the buffer and writing is done completely filling up the buffer, we switch them, now the SRAM which is full is used for reading and the SRAM which is empty is used for writing as shown in figure 4.21. In this way we can use two single port SRAM to design dual port SRAM which helps in achieving simultaneous reads and writes. We could not use dual port SRAM as it was consuming more than double the area and power for single port SRAMs. Coming to the working part as shown in figure 4.22. since the arrays were working in SIMD fashion, we had to keep a check on the address. And all the address would be read sequentially. Once we reach the maximum address all the bits would be high, and this turns our flag. We use two separate flag one for read and other for write. To account for asynchronous reads and write. Once both the flags are high, we swap our buffers.

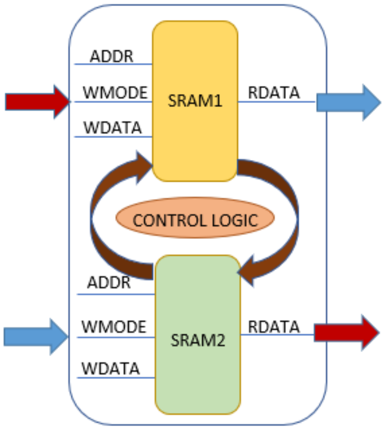


Figure 4.21. Double buffering high level diagram

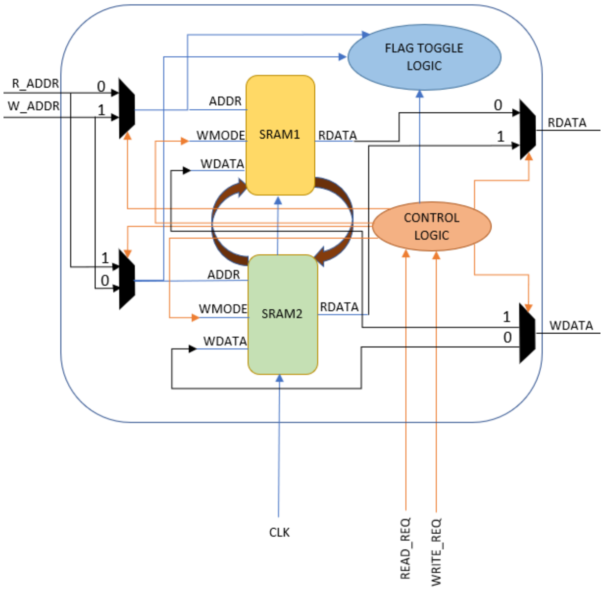


Figure 4.22. circuit level dataflow

Figure 4.23 shows the FPGA block diagram of scratchpads used in weight buffers. Implementation was done in FPGA for power, area analysis. From figure 4.24 and table 4.8 and 4.9 on comparing the delay area and power of dual port SRAM with that of double buffering, we have used a FPGA to compute the area and power. We see that double buffering uses 12% of the area in comparison to dual port which uses 34% area. All using double buffering we obtain power saving of 74%. 3.66W to 0.92W in case of double buffering. Critical path shown in table 4.6 remains the same. However, using single port SRAMs in double buffering configuration saves power.

Table 4.8. Scratchpad area utilization

Resource	Utilization	Available	Utilization %
LUT	5	53200	0.01
LUTRAM	3	17400	0.02
FF	3	106400	0.00
IO	11	125	8.80

Table 4.9. Dual port SRam area utilization

Resource	Utilization	Available	Utilization %
LUT	143	53200	0.27
LUTRAM	96	17400	0.55
FF	16	106400	0.02
IO	39	125	31.20

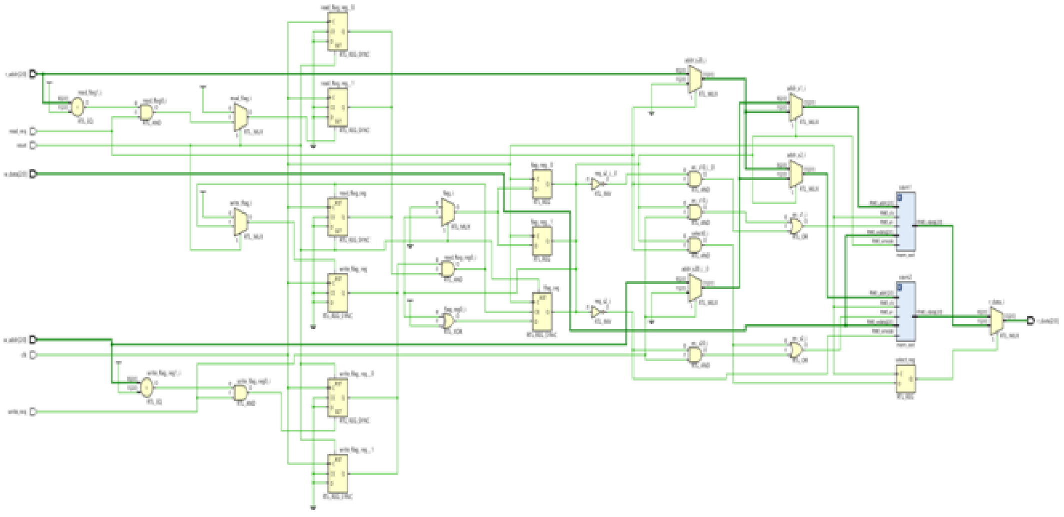
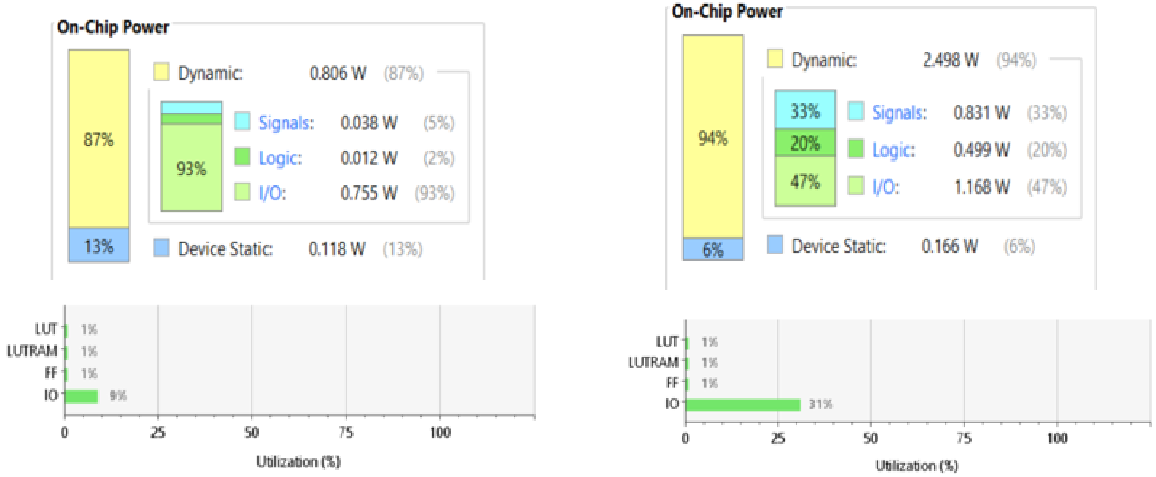


Figure 4.23. FPGA block diagram of double buffering scratchpad

4.8 Verification

The design has been verified properly to make sure it is reaching the expected design requirement. Most of the verification was automated to get rid of manual testing. For this purpose, we have used Cocotb. It is easier to write verification in software as we can more exactly try to imitate the real environment and is simpler in terms of coding. Cocotb is a simulator plugin for your RTL digital design simulator and has a python library that allows you to write code that is compatible with this plugin as shown in figure 4.25. We start with our RTL code which can be run on standard simulator. We have our test bench written in python. The link between the RTL simulator and python testbench is provided by using Cocotb. The link works by using VPI FLI, VHPI, which are the standard interfaces to talk with your simulator. High level testcases were created to make the design check end to end. The memory read/write betwwen weight buffers and PEs was tested. All arithmetic units were checked to make sure that PE which is mainly multiply and accumulate is working fine. Also the design was imported from TPU paper. Proper test cases were written to check the activation/partial sums flow and to see if it is matching TPU requirements. Bidirectional systolic arrays are needed for the design to flexibly



(a) Scratchpad

(b) Dualport SRAM

Figure 4.24. Power, Area comparison between double buffering config and dual port SRAM

fission into multiple DNN accelerators. So, the design needed to be properly checked for the bidirectional flow. And the connectivity was checked to see if any unpredictable errors would popup because of the fission connections. Finally all these changes were checked for single input and multiple inputs which involves piplining the inputs for maximum utilization.

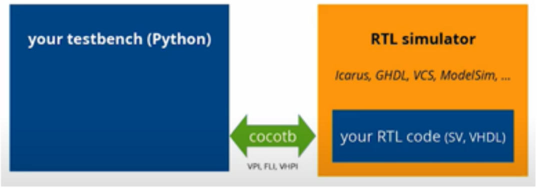


Figure 4.25. Cocotb high level block diagram

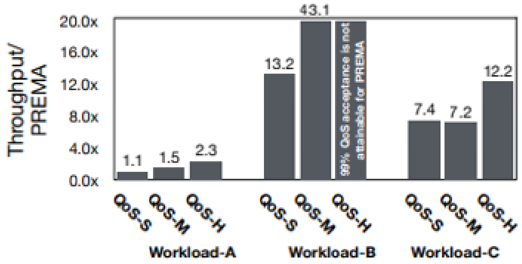


Figure 4.26. Planaria throughput improvement over PREMA. for different QoS latency constraints and workloads.

4.9 Experimental results

Initially we compare we PREMA. Throughput comparison. Figure 4.26 compares the throughput of Planaria with PREMA across various workload scenarios and QoS requirements. As described, we use a Poisson distribution (λ) to generate the arrival times for queries with a given throughput of (1λ). For both Planaria and PREMA, we report the maximum possible throughput (1λ) that meets the SLA, following the MLPerf methodology. For Workload-C as the most comprehensive workload scenario that encompasses all the benchmark DNNs, Planaria improves the throughput by 7.4×, 7.2×, 12.2×, for QoS-S, QoS-M, and QoS-H, respectively. For Workload-B the improvements increase to 13.2 ×, 43.1 × for QoS-S and QoS-M, while for the case of QoS-H, the baseline PREMA does not meet the 99% QoS constraints. This trend emanates from the fact that the DNNs in Workload-B include separable depth-wise/point-wise convolutions (except for Tiny YOLO). Since Planaria has fission capability, it can better utilize its resources for depth-wise convolution while a monolithic design in PREMA cannot conform to the requirements of this layer. This is an additional advantage of fission that enables running these recent DNNs more efficiently. About Workload-A, Planaria improves the throughput by 1.1, 1.5, 2.3, for QoS-S, QoS-M, QoS-H, respectively. These DNNs do not include depth-wise convolution, yet our hardware and scheduling yields significant benefits. Across all three workload scenarios, improvements are more significant for the case of hard QoS. Our task scheduling algorithm considers QoS along with other metrics that leads to better throughput for stricter QoS requirements.

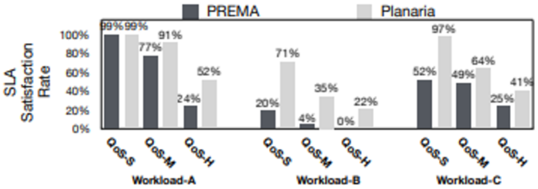


Figure 4.27. SLA satisfaction rate of Planaria and PREMA.

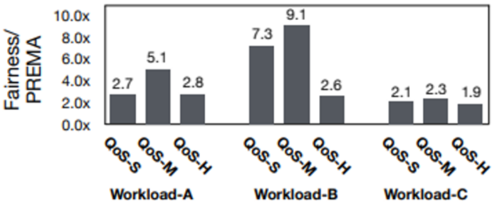


Figure 4.28. System fairness improvement normalized to PREMA.

SLA satisfaction rate comparison. Figure 4.27 illustrates the SLA satisfaction rate of Planaria and PREMA for the same throughput (1λ). As the results show, Planaria improves the SLA satisfaction rate across all the workloads and QoS requirements. The Planaria's fission-capable microarchitecture combined with its QoS-aware task scheduling algorithm enables significantly larger number of workloads to be executed while adhering to SLA, compared to PREMA.

Based on the adopted QoS constraints from [49], Workload-A allows relatively larger slack time compared to other workloads. As such, both Planaria and PREMA performs relatively better in SLA satisfaction for Workload-A. Except for the case of QoS-S, where both Planaria and PREMA satisfy the SLAs 99% of the time, Planaria provides a 14% and 28% increase in SLA satisfaction rate compared to PREMA. For the case Workload-B that requires tighter QoS as compared to Workload-A, improvements increase to 22%, 31%, and 51%, for QoS-H, QoS-M, and QoS-S, respectively. Finally, the improvements range from 16% to 45% for QoS-S to QoS-H, with respect to the mixed Workload-C. Fairness comparison. Figure 4.28 shows fairness with Planaria normalized to fairness with PREMA across all the three workload scenarios. Planaria significantly improves fairness for Workload-A by 2.8×, 5.1×, 2.7× across the three QoS requirements. Overall, Planaria improves fairness significantly with minimum of 1.9× for (Workload-C, QoS-H) and maximum of 9.1× for (Workload-B, QoS-M). That is because Planaria can spatially co-locate multiple tasks and process them simultaneously, as opposed to PREMA's monolithic accelerator and its preemptive approach that can only process one task at a time. In addition to that, Planaria's task scheduling algorithm (functions ALLOCATEFITTASKS and ALLOCATEUNFITTASKS in Algorithm 1) ensures that each dispatched task receives adequate number of subarrays with respect to its priority and overall execution time.

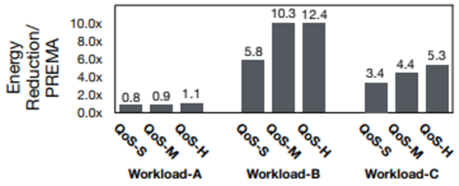


Figure 4.29. Planaria energy reduction compared to PREMA.

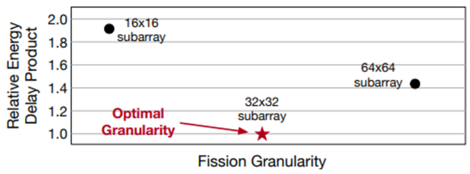


Figure 4.30. Design space exploration for fission granularity

Energy comparison. Figure 4.29 compares the total energy consumption for the execution of workloads on Planaria and PREMA systems. For Workload-A, Planaria consumes slightly more energy than PREMA ranging 11% (QoS-M) to 25% (QoS-S). Multi-tenancy leverages the slack in QoS requirements and as such runs the application slightly slower than an isolated mode to improve throughput and fairness. This slower execution manifest itself as increased total energy compared to running each DNN in isolation with fastest possible speed without considering QoS. As a result, we see a degree of total energy increase for these traditional workloads. In the case of Workload-B and Workload-C, however, when modern DNNs are mixed, the energy benefits from fission outweighs this effect. Workload-B enjoys the maximum energy improvements using Planaria, with minimum of 5.8× and maximum of 12.4× gain over PREMA. Planaria's fission-capable design enables it to adapt to the various computational demands that exist in this workload (e.g. EfficientNet-B0, MobileNet-v1) and significantly reduces their energy consumption. Overall, with respect to Workload-C which is a mixture of both DNN classes, Planaria reduces the total energy consumption of the workloads by 3.4, 4.4, and 5.3× for QoS-S, QoS-M and QoS-H, respectively. Scaling out resources. Figure 4.32 illustrates the minimum number of Planaria nodes necessary to achieve 99% SLA satisfaction. We use a constant throughput across all workloads and QoS requirement. In this scaled-out setting, the DNN task traffic is distributed across multiple Planaria-equipped node, where each node has one accelerator. Each single DNN task is not distributed across multiple nodes and is only mapped to one chip, while it can be co-located with multiple tasks on the same accelerator chip. As illustrated in the figure, the number of nodes necessary to achieve SLA satisfaction increases as we go from soft (QoS-S) to hard constraints (QoS-H) on QoS. Among the various workload scenarios, Workload-B that has stricter QoS constraints, requires larger number of nodes compared to other workloads, with the minimum of 2 nodes for QoS-S and maximum of 7 nodes for QoS-H. As also shown in Figure 4.27, Workload-A with QoS-S does not need increase in number of nodes and one Planaria accelerator is sufficient, while it requires 3 nodes with QoS-H. Finally, with regard to Workload-C, 2, 3, and 5 nodes are required for QoS-S, QoS-M, andQoS-H, respectively.

4.9.1 Sensitivity studies

Planaria performance/energy on a single DNN inference. Figure 4.31 shows the speedup and energy reduction of Planaria as compared to a conventional systolic-based accelerator (like PREMA's) with the same amount of compute and memory resources, while each DNN inference is executed in isolation. Across the nine DNN benchmarks, Planaria offers 3.5× and 6.4× speedup and energy reduction, respectively. The fission-capable design of Planaria enables it to adapt to the various computational characteristics that exist in DNN layers and exploits the opportunities for parallelism and data reuse to improve the performance and energy consumption. Among them, EfficientNet-B0, MobileNet-v1, and SSD-M that exploit depth-wise convolutions, enjoy the maximum benefits. In the case of depth-wise convolution layers, Planaria's dynamic fission capability enables it to decompose to a larger number of smaller systolic subarrays that operate in parallel. These subarrays process multiple filter channels in parallel, leading to higher utilization of PEs and consequently significant increase in performance. About DNNs without depth-wise convolution, Tiny YOLO enjoys the maximum benefits, 2.8× speedup and 5.7× energy reduction. GNMT enjoys the least improvements, since it mostly requires matrix-multiplication operations, which is also suitable for a monolithic design. Unlike the multi-tenant case for Workload A, there is no increase in energy for its isolated DNNs. As discussed, multi-tenancy trades off individual energy and speed for higher throughput. In the isolated case, that tradeoff is not employed, and all the resources are allocated to one DNN maximizing its efficiency and speed through fission.

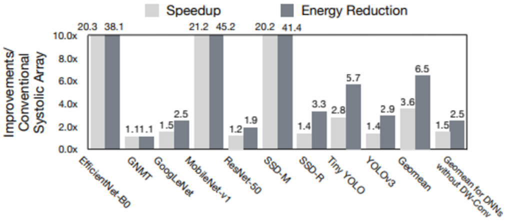


Figure 4.31. Speedup and energy reduction for single DNN inference in Planaria compared to a conventional accelerator.

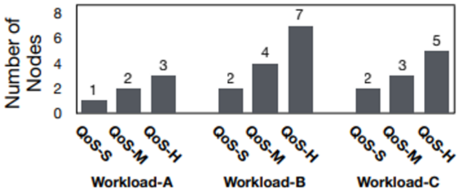


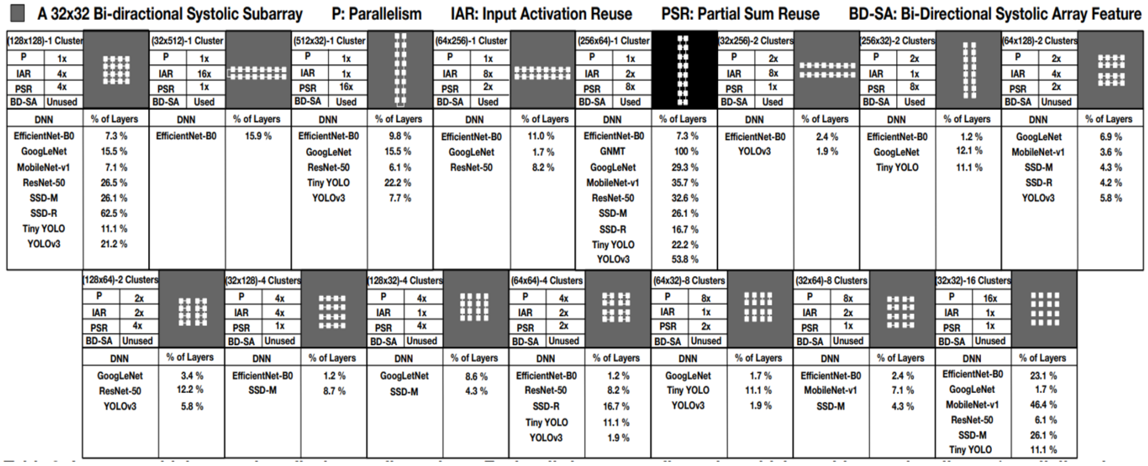
Figure 4.32. Required number of nodes to achieve 99% SLA satisfaction.

Design space exploration for fission granularity. To find the optimal fission granularity, we perform a design space exploration that yields the most efficient granularity, as shown in Figure 4.30. We consider 128 × 128 total number of PEs (as was in PREMA and TPU) and sweep the size of subarrays for 16 × 16, 32 × 32, and 64 × 64. To find the optimal size, we consider Energy-Delay-Product (EDP) and measure its average value across the benchmarked DNNs, while they run in isolation. Figure 4.30 illustrates the relative EDP values for the three design points. As also shown in Figure 4.30, 32 × 32 offers least EDP and as such we use this size as fission granularity for Planaria. Sensitivity analysis for fission possibilities. Table 4.10 illustrates the sensitivity of DNN layers to various fission possibilities. For this analysis, we considered the nine benchmark DNNs in an isolated setting, where the whole accelerator is dedicated to a single DNN inference. The gray cells of the table show the 15 fission possibilities that are determined most fitting for the benchmark DNNs when run in isolation. The table also reports their architectural characteristics (parallelism (P), input activation reuse (IAR), partial sum reuse (PSR), and usage of bi-directional systolic movement in that configuration) with respect to the 3232 fission granularity. A cell also lists the DNNs with the percentage of their layers that have utilized the pertinent fission configuration. Six of these 15 configuration are enabled through Bi-directional systolic design, providing architecture fission points that are beneficial for DNNs. The black cell in Table 4.10 captures the most prevalent and fruitful fission configuration across the benchmarks that, in fact, exploits the bi-directional feature. All nine DNNs utilizes this configuration in their execution, while GNMT,

YOLOv3, and MobileNet-v1 are the three DNNs that utilizes this configuration more than other DNNs. Another important configuration is where fission takes place at the finest granularity and 16 number of 3232 subarrays work independently in parallel. This configuration is specifically important and useful for DNNs with depth-wise convolution, namely EfficientNet-B0, MobielNet-v1, and SSD-M, where each subarray processes a channel, leading to higher utilization compared to monolithic systolic array. Area and power overheads for fission.

Figure 4.33 illustrates the breakdown of area and power with respect to different hardware

Table 4.10. Layer sensitivity to various fission configurations. Each cell shows a configuration with its architectural attributes and the percentage of the layers that uses the configuration



▨ A 32x32 Bi-diractional Systolic Subarray P: Parallelism IAR: Input Activation Reuse PSR: Partial Sum Reuse BD-SA: Bi-Directional Systolic Array Feature

Configuration	P	IAR	PSR	BD-SA	DNN	% of Layers
(128x128)-1 Cluster	1x	4x	4x	Unused	EfficientNet-B0 GoogLeNet MobileNet-v1 ResNet-50 SSD-M SSD-R Tiny YOLO YOLOv3	7.3 % 15.5 % 7.1 % 26.5 % 26.1 % 62.5 % 11.1 % 21.2 %
(32x512)-1 Cluster	1x	16x	1x	Used	EfficientNet-B0	15.9 %
(512x32)-1 Cluster	1x	1x	16x	Used	EfficientNet-B0 GoogLeNet ResNet-50 Tiny YOLO YOLOv3	9.8 % 15.5 % 6.1 % 22.2 % 7.7 %
(64x256)-1 Cluster	1x	8x	2x	Used	EfficientNet-B0 GoogLeNet ResNet-50	11.0 % 1.7 % 8.2 %
(256x64)-1 Cluster	1x	2x	8x	Used	EfficientNet-B0 GNMT GoogLeNet MobileNet-v1 ResNet-50 SSD-M SSD-R Tiny YOLO YOLOv3	7.3 % 100 % 29.3 % 35.7 % 32.6 % 26.1 % 16.7 % 22.2 % 53.8 %
(32x256)-2 Clusters	2x	8x	1x	Used	EfficientNet-B0 YOLOv3	2.4 % 1.9 %
(256x32)-2 Clusters	2x	1x	8x	Used	EfficientNet-B0 GoogLeNet Tiny YOLO	1.2 % 12.1 % 11.1 %
(64x128)-2 Clusters	2x	4x	2x	Unused	GoogLeNet MobileNet-v1 SSD-M SSD-R YOLOv3	6.9 % 3.6 % 4.3 % 4.2 % 5.8 %
(128x64)-2 Clusters	2x	2x	4x	Unused	GoogLeNet ResNet-50 YOLOv3	3.4 % 12.2 % 5.8 %
(32x128)-4 Clusters	4x	4x	1x	Unused	EfficientNet-B0 SSD-M	1.2 % 8.7 %
(128x32)-4 Clusters	4x	1x	4x	Unused	GoogLetNet SSD-M	8.6 % 4.3 %
(64x64)-4 Clusters	4x	2x	2x	Unused	EfficientNet-B0 ResNet-50 SSD-R Tiny YOLO YOLOv3	1.2 % 8.2 % 16.7 % 11.1 % 1.9 %
(64x32)-8 Clusters	8x	1x	2x	Unused	GoogLeNet Tiny YOLO YOLOv3	1.7 % 11.1 % 1.9 %
(32x64)-8 Clusters	8x	2x	1x	Unused	EfficientNet-B0 MobileNet-v1 SSD-M	2.4 % 7.1 % 4.3 %
(32x32)-16 Clusters	16x	1x	1x	Unused	EfficientNet-B0 GoogLeNet MobileNet-v1 ResNet-50 SSD-M Tiny YOLO	23.1 % 1.7 % 46.4 % 6.1 % 26.1 % 11.1 %

components in Planaria. when synthesized at 45 nm. This break down does not include the buffers that amounts 12 MB and is the same as the one used in PREMA. The breakdown includes the components added to support dynamic fission, which includes, logic for bi-directional flow of data, Fission Pod crossbar, SIMD vector unit additions, instruction buffer additions, re-configurations registers overheads. Overall, dynamic fission adds 12.3%, 20.3% extra area and power, respectively.

4.10 Related Work

The need for higher speed and efficiency in DNN execution has led to an explosion of DNN accelerators that has even made their way to operational datacenters (Google's TPU , NVIDIA T4, Microsoft Brainwave, etc.). However, multi-tenancy has been largely omitted in the proposed or deployed designs due to the arms race in the market for higher speed and efficiency. Even the MLPerf benchmark suite keeps this single model focus for both training and inference. In contrast, this paper offers spatial multi-tenant acceleration through architecture fission that is propelled by unique microarchitectural mechanisms and organizations that enables flexible task scheduling. As such, this paper lies at the intersection of DNN acceleration and multi-tenant execution. We discuss relevant related work categories below. Multi-tenancy for DNN accelerators. PREMA

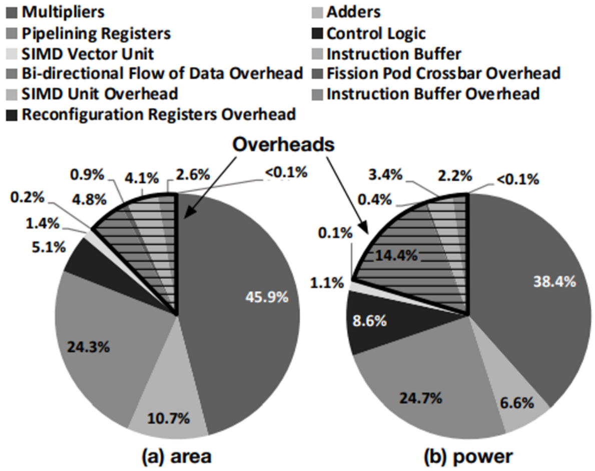


Figure 4.33. Planaria power/area breakdown and its overheads when synthesized with 45 nm technology.

develops a scheduling algorithm for preemptive execution of DNNs on a monolithic accelerator and uses time-sharing for multitenancy. In contrast to this temporal multi-tenancy scheduling framework, this paper explores architecture design for spatial co-location of DNNs for multi-tenant acceleration and its unique scheduling challenges. A concurrent work in the ISCA 2020 website aims to support simultaneous multi-neural network execution. However, the paper is neither published nor is available for qualitative or quantitative comparison. Flexibility in DNN accelerators. Flexibility in DNN acceleration has been recently gained attention. However, these inspiring works do not explore simultaneous spatial co-location of multiple DNNs on the same chip. Eyeriss v2 proposes a hierarchical architecture equipped with a flexible mesh based NoC that provides flexibility to adapt to various level of data reuse. MAERI and SIGMA propose a reconfigurable interconnect among the PEs to deal with sparsity in neural networks and matrix multiplication. Simba proposes a scalable multi-chip module-based accelerator to reduce fabrication cost and provide scalability with respect to inter-chip and intra-chip communication. Bitfusion explores bit-level dynamic composability in its multipliers to support heterogeneity in deeply quantized neural networks. Tangram explores dataflow optimizations by buffer-sharing dataflow and inter-layer pipelining on a hierarchical design to reduce energy. These works do not explore multi-tenancy nor simultaneous co-location of multiple DNNs. Multi-tenancy for CPUs and GPUs. There is a large swath of related work on multi-tenancy for CPUs and GPUs due to its vitality for cloud-scale computing. NVIDIA Triton Inference Server provides a cloud software inference solution optimized for GPUs and offers benefits by supporting multi-tenant execution of DNNs on them. Grand SLAM proposes scheduling policies to minimize SLA violation rates for microservices at the cloud for CPUs and GPUs. The studied workloads include DNNs. In contrast, this paper uniquely enables spatial multi-tenancy on DNN accelerators, by leveraging a dynamic fission in the architecture and leveraging that through the scheduler. Kubernetes and Mesos are cloud-scale resource management framework and, due to the unavailability of spatial multi-tenancy in DNN accelerators, have not explored that aspect of scheduling. Our scheduling algorithm is complementary to their operation. DNN acceleration. There is a large body of work for isolated acceleration of DNNs that, although inspired our work, are not focused on multi-tenancy, and rather offered various innovations to improve the speed and efficiency of DNN execution.

4.11 Conclusion

As inference-as-a-Service is growing in demand, it is timely to explore multi-tenancy for DNN accelerators. This paper explored this topic through a novel approach of dynamic architecture fission, and provided a concrete architecture, Planaria, and its respective scheduling algorithms. Evaluation with a diverse set of DNN benchmarks and workload scenarios shows significant gains in throughput, SLA satisfaction, and fairness.

.

Chapter 4, in part, has been submitted for publication of the material as it may appear in MICRO 2020. The thesis author is one of the co-author in this paper. This paper ’PLANARIA’ is coauthored with G. Soroush, A. Byung Hoon, K. Joon Kyung, K. Sean, Y. Brahmendra, S. Hardik, M. Alian, k. Nam Sung, E. Hadi. I would like to thank them for allowing me to be a contributor for their work.

Chapter 5

Accelerating Training Phase

5.1 Introduction

Training DNNs with large datasets takes a significant amount of time and is a major bottleneck in the field of deep learning. To accelerate the training phase, many GPUs are deployed to mitigate the runtime from many months to many weeks. In addition, the long runtime of deep learning algorithms avoids a comprehensive design space exploration. Therefore, the training phase is a good candidate for hardware acceleration.

In this section, we review the architectural requirements of such accelerators. We discuss these challenges and review the impact of addressing them on the design of a training accelerator based on in-situ computing. It is worth noting that our analysis in this section is limited to MLPs and CNN-based networks. For other deep networks such as RBMs (Restricted Boltzmann Machines) that have different structure and low-resolution weights, in-situ computing might lead to different results. Many factors come into picture while off loading data like the network topology, amount of data being off loaded and the type of training algorithm used. In the following section we discuss few acceleration techniques for training as well as for off loading resources to remote data centre and see how they can be improved. Distributed training has been a major driver for advances in DNN by reducing the training time. Although distributing training improves the compute power, it comes with cost of inter node communication proportional to the DNN size[31]. In general, as shown in figure 5.1 a distributed training system are structed as a hierarchy of worker aggregator nodes.

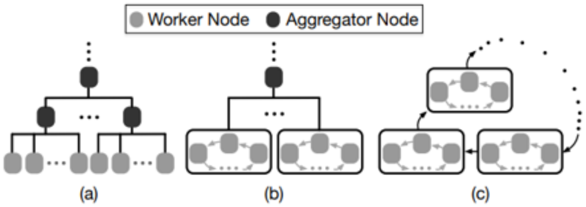


Figure 5.1. (a) SOTA hierarchical distributed training. (b) INCEPTIONN's distributed training. (c) Hierarchical use of INCEPTIONN.

In each iteration the aggregator nodes gather the gradient updates from their sub nodes, communicate the cumulative gradients upwards and send back the updated weights downwards. This movement of information imposes load on the network and this cost can be reduced by embedding data compression accelerators in the Network Interface Cards. Using compression techniques and in-network accelerators for the compression provides limited gain due to complexity and latency overhead. So, INCEPTIONN is used, that offers a novel gradient compression technique, its in-network accelerator architecture, and a gradient-centric distributed training algorithm to maximize the benefits of the in-network acceleration. Gradients in contrast to weights are more amenable to precision can be compressed with less loss. The existing training algorithms communicate gradients in only one leg of the communication, which reduces the opportunities for compression and its in-network acceleration. Using compression can make the aggregators the bottleneck. Building on the above three observations INCEPTIONN comes up with a lossy-compression algorithm for floating-point gradient values. This compression exploits a unique value characteristic of gradients which has values between -1.0 and 1.0 and the distribution peaks around zero with low variance.

To favor gradients and reduce burden of multiple stream it uses gradient-centric, aggregator-free training algorithm which leverages both legs as shown in figure 5.1. This algorithm enables the distributed nodes to only communicate gradients and equally share the load of aggregation, which provides more opportunities for compressing gradients and improved load balance among the nodes. And the partial aggregates are sent from one node to another is a circular fashion.

This combination of lossy compression algorithm for gradients, NIC integrated compressed accelerator and gradient centric aggregator free training algorithm is combined to alleviate the communication bottleneck without affecting the mathematics of DNN training.

5.2 Distributed training for deep neural networks

DNN training involves determining weights w and using input data x and yields a prediction y. Supervised training finds w by minimizing a loss function ‘(F(x,w),y *). DNNs are commonly optimized using gradient descent, which updates the weights in the opposite direction of the loss function’s gradient. The weights are updated using the following rule

$$\begin{gathered} w^{(t+1)} = w^{(t)} - \eta \cdot \frac{\partial l_D}{\partial w^{(t)}} = w^{(t)} - \eta \cdot g^{(t)} \\ w^{(t+1)} = w^{(t)} - \eta \cdot \sum_i \frac{\partial l_{B_i^{(t)}}}{\partial w^{(t)}} = w^{(t)} - \eta \cdot \sum_i g_i^{(t)} \end{gathered} \tag{5.1}$$

where $w(t+1)$, $w(t)$, and $g(t)$ denote the next updated weights, the current weights, and the current gradient, respectively. The parameter is the learning rate. However, contemporary datasets D do not fit into the memory of a single computer and this is handled using stochastic gradient descent where we sample a subset or minibatch B from D. To parallelize the D is divided into partial sets D_i and assigned to worker nodes n_i which in turn divides it into minibatch B_i to calculate partial gradient which are updated. The aggregator node, then, can send back the updated weights $w(t+1)$ to all worker nodes. This mathematical formulation avoids moving the training data and only communicates the weights and gradients.

as illustrated 5.2, In these algorithms, worker and aggregator nodes construct a tree where the leaves are the worker nodes that compute the local gradient $(g(t)i)$ and the non-leaf nodes are the aggregator nodes that collect the calculated local gradients to update the weights $(w(t))$ and send back the updated weights $(w(t+1))$ to worker nodes. However, each aggregator node should communicate with a group of worker nodes and aggregate the local gradients, which

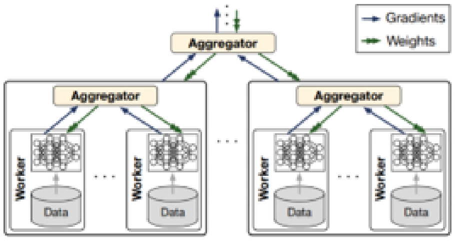


Figure 5.2. Worker-aggregator approach for distributed training.

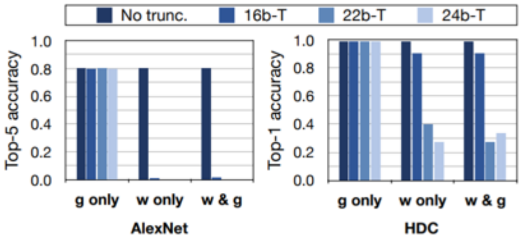


Figure 5.3. Impact of floating-point truncation of weights, gradients on training accuracy.

becomes the communication and computation bottleneck.

In the above figure 5.3 we see the exchanged $weight/gradient\ \ size$ and the fraction of communication time when training DNN models. The $communication/computation\ \ ratio$ becomes even larger as the specialized accelerators deliver higher performance and reduces the computation time and/or more nodes are used for training. To reduce the communication overhead, INCEPTIONN aims to develop a compression accelerator in NICs. Leveraging the three observations motivates for the design of our lossy compression for gradients. Both weights and gradients in distributed training are normally 32-bit floating-point values, whereas they are 16 or 32-bit fixed-point values in the inference phase. Floating-point values are not very much compressible with lossless compression algorithms. Thus, we employ a more aggressive lossy compression, exploiting tolerance of DNN training to imprecise values at the algorithm level.

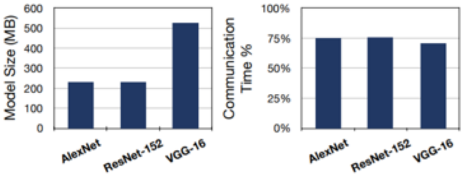


Figure 5.4. (a) The size of w/g. (b) % of the time spent to exchange g and w in total training time with a conventional worker-aggregator approach.

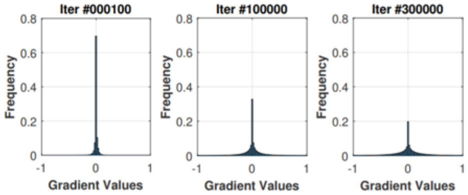
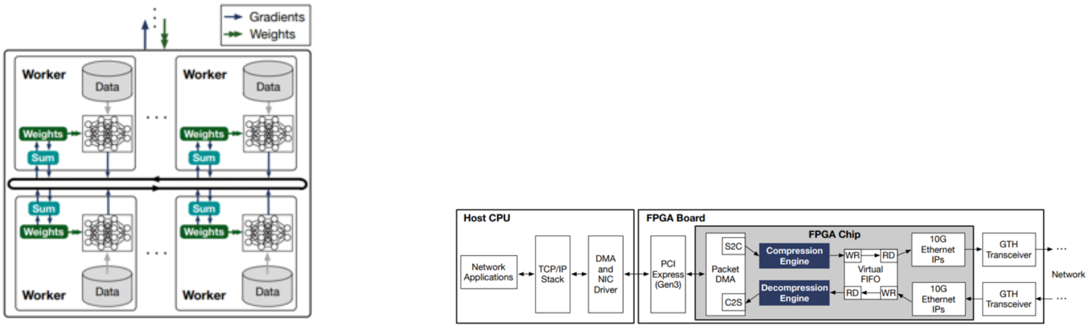


Figure 5.5. Distribution of AlexNet gradient values at early, middle, and final training stages.

This result from figure 5.4 shows that the truncation of g affects the predictor accuracy significantly less than that of w, and the aggressive truncation of w detrimentally affects the accuracy for complex DNNs as precision loss of w is accumulated over iterations while that of g is not. As shown in figure 5.5 all the gradient values are between -1 and 1 throughout the three training phases and most values are close to 0. Given this, we focus on the compression of floating-point values in this range to minimize the precision loss.

5.3 Gradient centric distributed training



(a) Worker group organization.

(b) An example of distributed gradient exchange.

Figure 5.6. INCEPTIONN gradient-centric distributed training algorithm in a worker group.

The figure 5.6 illustrates the worker group organization of the INCEPTIONN training algorithm. In this algorithm, there is no designated aggregator node in the worker group. Each worker node maintains its own model w, and only exchanges and aggregates a subset of gradients g with two neighboring nodes after each iteration. Following figure illustrates the algorithm step by step. Initially, every worker node starts with the same w0 and INCEPTIONN evenly partitions gradient vectors into four blocks, for four worker nodes. Every training iteration each node loads and computes a mini batch of data based on the current w and then generates a local g to be exchanged.

Subsequently, INCEPTIONN exchanges and aggregates g in two phases. (P1) aggregation of gradients. $worker[0]$ sends $blk[0]$ to its next node, $worker[1]$. $Worker[1]$ performs a sum-reduction on the received $blk[0]$ and its own $blk[0]$ which happens concurrently across four workers. This step is repeated two more times until all workers have fully aggregated all blocks. (P2) propagation of the aggregated gradients. $worker[3]$ sends $blk[0]$ to $worker[0]$ resulting in two fully aggregated blocks which happens concurrently across all workers. This step is repeated until every worker has g which is fully aggregated from all four workers. In summary, the INCEPTIONN training algorithm utilizes the network bandwidth of every worker evenly unlike the worker aggregator approach, creating the communication bottleneck.

5.4 In network acceleration of gradient compression

Even a simple lossy truncation operation significantly increases the computation time, by packing/ unpacking many g values and burdens the CPUs which negates the benefit of reduced communication, decreasing the total training time. Therefore, to reduce both communication and computation times, we need hardware-based compression for INCEPTIONN.

As shown figure 5.7-5.9, we insert the accelerators within the NIC reference design and integrate the compression and decompression engines. For output traffic, the packet DMA collects the network data from the host system through the PCIe link and goes through the Compression Engine that stores the resulting compressed data in the virtual FIFOs that are used by the 10G Ethernet MACs. These MACs drive the Ethernet PHYs on the board and send or receive the data over the network. For input traffic, the Ethernet MACs store the received data from the PHYs in the virtual FIFOs. Once a packet is complete, the Decompression Engine starts processing and passing it to the packet DMA for transfer to the CPU. Both engines in figure 5.7, 5.8 use the standard 256-bit AXI-stream bus to interact with other modules. These algorithms process streams of floating-point numbers, while the NIC deals with TCP/IP packets and accelerators need to be customized for it. NIC needs to provide the abstraction that enables the software to activate/deactivate the lossy compression per packet basis. Compression Engine first needs to identify which packets are intended for lossy compression. Then, extract payload, compress it, and then reattach it to the packet. It processes packets in bursts of 256 bits, as an AXI interface can deliver in one cycle. API marks a packet compressible by setting the Type of Service field in the header to a special value. If ToS value mismatches, compression is bypassed.

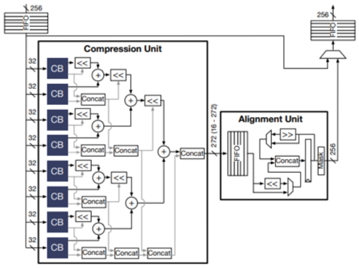


Figure 5.7. 256-bit burst compressor architecture.

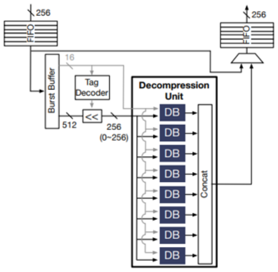


Figure 5.8. 256-bit burst decompressor architecture

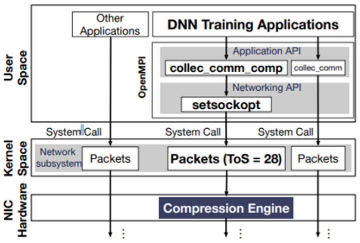


Figure 5.9. Dataflow across the software stack and NIC hardware.

As shown the payload burst feeds into the Compression Unit equipped with eight Compression Blocks (CBs), each of which performs the compression described. Each CB produces a variable-size output in the size of either 32, 16, 8, or 0 bits, which need to be aligned as a single bit vector. Binary shifter tree produces the aligned bit vector. The 2-bit tags of the eight CBs are simply concatenated as a 16-bit vector. Finally, the aligned bit vector and tag bit vector are concatenated as the final output of the Compression Unit. For each burst, a variable-size (16 – 272) bit vector needs to be aligned so that we can transfer the 256-bit burst via the AXI interface and is done by the alignment unit. Using ToS field the decompression unit needs to identify the compressed packets. The compressed burst has 8 FP numbers which overlap two consecutive bursts could be insufficient to proceed to the decompression. Therefore, a Burst Buffer that maintains up to two bursts is present. Once filled it feeds the 16-bit tag to the Tag Decoder to calculate the size of the eight compressed bit vectors which along with tag bit vectors are fed into the eight Decompression Blocks, which executes algorithm, concatenates and transfer via AXI interface. For the next cycle, Burst Buffer shifts away the consumed bits and reads the next burst.

Packets that need to be modified are tagged with a reserved value of 0x28. When we co-run DNN training application and some other networking applications on a server MPI collective communication comp is used to tag TCP/IP packets. MPI collective communication comp propagates a variable down to the OpenMPI networking APIs and sets the ToS option of the

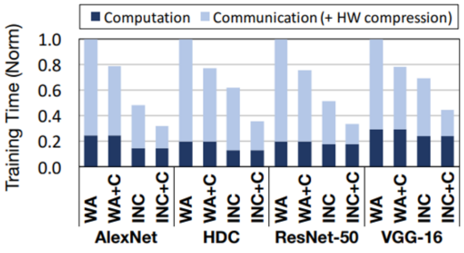
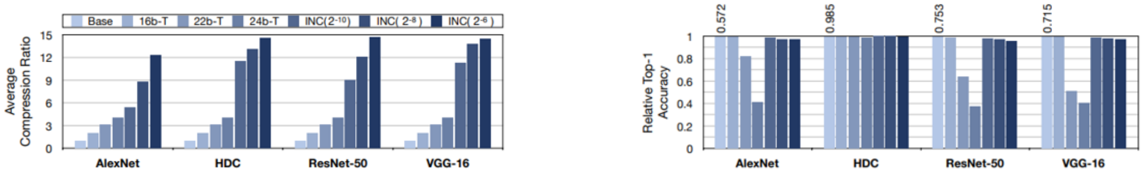


Figure 5.10. Training time between the worker-aggregator based approach and the INCEPTIONN with and without hardware-based compression in NICs.

corresponding TCP sockets used for communication. Based on the value, comparator selects to use or bypass the compression engines.

5.5 Evaluation

Irrespective of DNN model less than 30% time is spent for local computation and 70% of time is spent for communication. Graph from figure 5.10 shows that even in a small cluster without compression, the INCEPTIONN's training algorithm offers shorter total training time. The concurrent utilization of all the links among nodes makes it efficient. Also balanced gradient exchange contributes to the reduction of computation time as the gradient summation is done by all the nodes in a distributed manner. The Inceptionn with gradient compression provides 80.7% and 53.9% lower communication time than the work aggregator and Inceptionn baseline providing a 2.2 – 3.1X speed up. The accuracy loss due to compression as shown in figure 5.11 might affect the final accuracy or prolong training time. On measuring only a modest number of epochs are needed to achieve similar accuracy and still offers the same speed up.



(a)

(b)

Figure 5.11. (a) compression ratios and (b) impacts on prediction accuracy of DNNs trained by INCEPTIONN training with lossy compression schemes.)

The naïve truncation of fp values provides low constant compression ratios while suffering from accuracy loss as errors here are uncontrollable and open ended. Also, the truncation is limited by mantissa where dropping more bits will perturb the exponent. In contrast the lossy compression provides higher compression ratios(15X) and preserves the training quality $(error < 2\%)$. The compression ratio of the gradients is not necessarily proportional to the reduction in communication time as we do not reduce the total number of packets and the network stack overhead such as sending network packet headers remains the same.

Gradient exchange time was measured for scalability. It consists of both gradient/weight communication and gradient summation time, and represents the metric in the scalability evaluation, because only communication and summation overheads scale with the number of nodes, while the time consumed by other DNN training steps such as forward pass, backward pass, weight update are constant due to their local computation nature. The gradient exchange time increases almost linearly with the number of worker nodes in the WA cluster; however, it remains almost constant in the INCEPTIONN cluster. As in WA the communication and summation loads congest the aggregator node, while the INCEPTIONN approach balances these two loads.

5.6 Conclusion

Communication is a significant bottleneck in distributed training. The community has pushed forward to address this challenge by offering algorithmic innovations and employing the higher speed networking fabric. However, there has been a lack of solution that conjointly considers these aspects and provides an interconnection infrastructure tailored for distributed training. This handled by INCEPTIONN by using in-network accelerator for the lossy compression of gradients and by using gradient centric distributed training. The communication time by reduced by $70.9 \sim 80.7\%$ and offers $2.2 \sim 3.1x$ speedup over the conventional training system, while achieving the same level of accuracy.

Chapter 6

Future Work

I believe that there are many important possible direction of research for expanding the multi tenancy property of DNN accelerators. The main idea is to split the monolithic systolic array into multiple parts to execute different DNN inferences. The task scheduler helps to prioritize the important work thereby increasing inferences. This helps improve the throughput a lot. Few state of the art technologies can be used to improve the power and flexibility of our design.

Accelerators used by Planaria still consume almost similar amount of power. New methodologies can be used to improve the power consumption of this systolic arrays. Like Minerva automates the design of DNN accelerators that achieve minimum power consumption while maintaining high prediction accuracy. This automated method if we are able to combine with Planaria will help achieve subsequent increase in throughput along with automatic deployment of low power DNN accelerators. The methods are discussed in the coming sections.

Also currently Planaria is only limited to DNN accelerators as it uses googles TPU architecture. The flexibility of this can be improved by incorporating TABLA which is a template based framework to accelerate the design phase of accelerators. The main idea that could be implemented is to use the automated flow to bring up different types of accelerators on planaria itself to exploit both dynamic allocation of accelerators along with automated design flow to improve its flexibility across multiple machine learning algorithms. This method are discussed in the following sections.

There could be many novel methods like bit level slicing, sparse networks, analog modelling. However, these two main ideas could have potential impact and change the performance and deployment of hardware accelerators. In following section we just briefly see how the designs are individually designed and leave to the future work for the integration part.

6.1 Low-Power, Highly Accurate Deep Neural Network Accelerators

6.1.1 Introduction

Deep neural networks are gaining popularity for solving a wide range of problems, from datacenters down to battery powered mobile and IOT devices. DNNs have become famous because of availability of massive dataset, access to highly parallel computational resources and improved algorithms. Through training we obtain the parameters that are fitted to data. While training is one-time cost, we need to perform inference continuously and this needs to be optimized for efficiency and power[30]. So, this paper tries to automate the design of DNN accelerators that achieve minimum power consumption while maintaining high prediction accuracy.

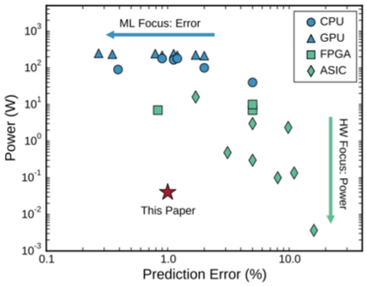


Figure 6.1. Survey reveals the disconnect between ML research and designing DNN accelerators.

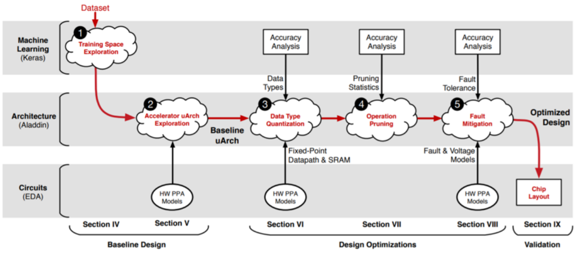


Figure 6.2. The five stages of Minerva. Analysis details for each stage and the tool-chain are presented

The following figure 6.1 shows values of MNIST prediction error and corresponding power consumption for different neural network implementations. The ML algorithms try to minimize the error, while the hardware community focusses on reducing the error. Current generation mobile devices already exploit DNN techniques across a range of applications. However, they typically offload computation to backend servers. This might pose problems of latency, autonomy, power consumption, and security for the growing number of applications. Rendering offloading is impractical pushing towards novel power reduction techniques.

Minerva, a highly automated codesign flow that combines insights and techniques across the algorithm, architecture, and circuit layers, enabling low power accelerators for executing highly accurate DNNs as shown in figure 6.2. Its flow first establishes a fair baseline design by exploring the DNN training and accelerator micro architectural design spaces, identifying an ideal DNN topology, set of weights, and accelerator implementation. After these three cross-layer optimization steps are applied to this baseline design: fine-grain, heterogeneous datatype quantization, dynamic operation pruning, and algorithm-aware fault mitigation for low-voltage SRAM operation. Implementing the above provides 8X power reduction compared to baseline.

6.1.2 Overview

Minerva consists of five stages : Stages 1–2 establish a fair baseline accelerator implementation. Stage 1 generates the baseline DNN: fixing a network topology and a set of trained weights. Stage 2 selects an optimal baseline accelerator implementation. Stages 3– 5 employ novel co-design optimizations to minimize power consumption over the baseline in the following ways: Stage 3 analyzes the dynamic range of all DNN signals and reduces slack in data type precision. Stage 4 exploits observed network sparsity to minimize data accesses and MAC operations. Stage 5 introduces a novel fault mitigation technique, which allows for aggressive SRAM supply voltage reduction. For each of the optimization stages ML level measures the impact of prediction accuracy, while architecture level evaluates hardware resource savings, and the circuit level characterizes the hardware models and validates simulation results.

The organization of five stages in 6.2 is done to minimize the possibility of compounding prediction error degradation. Initially training space exploration is done. Minerva first establishes a fair DNN baseline that achieves prediction accuracy comparable to state-of-the-art ML results.

This stage leverages the Keras software library to sweep the large DNN hyperparameter space. Of the thousands of uniquely trained DNNs, Minerva selects the network topology that minimizes error with reasonable resource requirements. The optimal network from Stage 1 is then fed to a second stage that thoroughly explores the accelerator design space. This process exposes hardware resource trade-offs through micro architectural parameters. Minerva then uses an optimal design point as the baseline to compare against. Minerva optimizes DNN data types with linear quantization analysis, independently tuning the range and precision of each DNN signal at each network layer. Quantization analysis minimizes bit widths without exceeding a strict prediction error bound. Compared to a 16-bit fixed-point baseline, data type quantization reduces power consumption by 1.5. The DNN kernel mostly comprises repeated weight reads and MAC operations. Analysis of neuron activity values reveals many operands are close to zero. Minerva identifies these neuron activities and removes them from the prediction computation such that model accuracy is not affected. Selective pruning further reduces power consumption by 2.0 on top of bitwidth quantization. By combining inherent algorithmic redundancy with low overhead fault mitigation techniques, optimization Stage 5 saves an additional 2.7 power by aggressively scaling SRAM supply voltages. Minerva employs state-of-the-art circuits to identify potential SRAM read faults and proposes new mitigation techniques based on rounding faulty weights towards zero. Minerva's optimizations reduce power consumption by more than 8 without degrading prediction accuracy.

6.1.3 Architecture

Stage 1 of Minerva explores the DNN training space, identifying hyperparameters that provide optimal predictive capabilities. To explore this configuration space, Minerva considers the number of hidden layers, number of nodes per layer, and L1/L2 weight regularization penalties. Minerva then trains a DNN for each point and selects the one with the lowest prediction error.Figure 6.3 shows that larger networks often have smaller predictive error which eventually saturate.

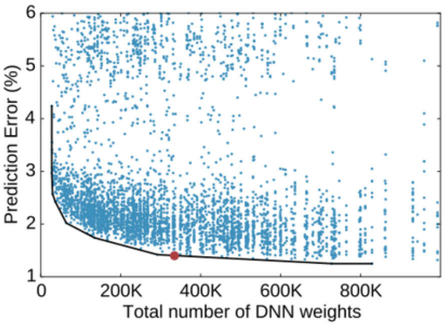


Figure 6.3. The black line indicates the Pareto frontier, minimizing DNN weights and prediction error. The red dot indicates the chosen network.

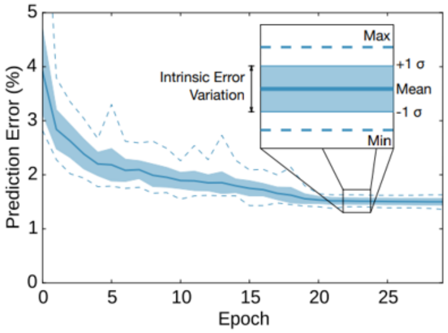
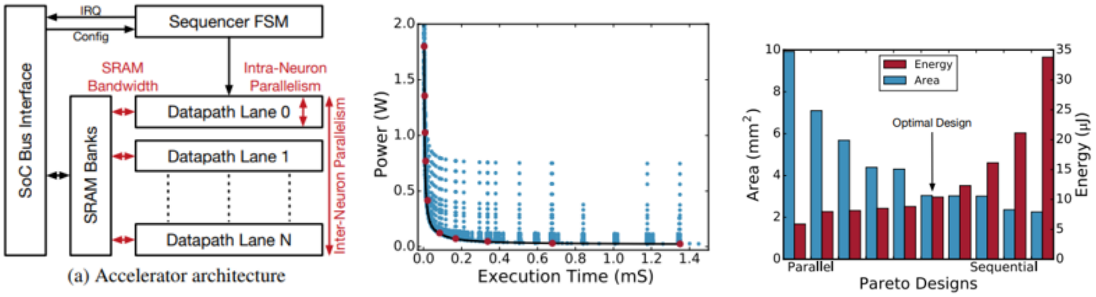


Figure 6.4. Intrinsic error variation is measured using random initial conditions and optimized design is maintained accuracy degradation below this.

Minerva modifies the calculations performed by the original DNN to optimize power and chip area for the resulting hardware accelerator which comes with increase in prediction error. To maintain DNN accuracy, we constrain the cumulative error increase from all Minerva optimizations to be smaller than the intrinsic variation of the training process. This interval is not deterministic, but sensitive to randomness from both the initialization of the pre-training weights and the stochastic gradient descent (SGD) algorithm. Figure 6.3,6.4 shows the average prediction error and a corresponding confidence interval, denoted by 1 standard deviation, obtained across 50 unique training runs. We use these confidence intervals to determine the acceptable upper bound on prediction error increase due to Minerva optimizations. For MNIST, the interval is 0.14%.

Stage 2 of Minerva takes the DNN topology and searches the microarchitectural design space for a superior DNN accelerator by generating and evaluating thousands of unique implementations using Aladdin and ultimately yields a power-performance Pareto frontier. We select a baseline design from this frontier that balances area and energy and then apply all remaining optimizations. The accelerator consists of memories for input vectors, weights, and activities as well as multiple data path lanes that perform neuron computations. Figure 6.6 below shows the layout of a single data path lane, consisting of two operand fetch stages F1 and F2, a MAC stage M, a linear rectifier activation function unit A, and an activation writeback WB stage.



(a) Accelerator architecture

Figure 6.5. High-level architecture,from a DSE over the accelerator implementation space, and energy and area analysis of resulting Pareto frontier designs.

The parts in black are optimized by Aladdin to consider different combinations of intra-neuron parallelism, internal SRAM bandwidth for weights and activities, and the number of parallel MAC operations. These features, in addition to inter-neuron parallelism collectively describe a single design point in the design space shown in Figure 6.6 . The red denotes additional logic needed to accommodate the optimizations described. The area and energy consumed by each of these Pareto design points is further shown in Figure. DNNs become parallel with memory bandwidths being the bottleneck, which is resolved by heavily partitioning SRAM into smaller memories. However, excessive scaling costs higher area at lower energy improvements. Under these constraints, the chosen design maximizes performance and minimizes power.

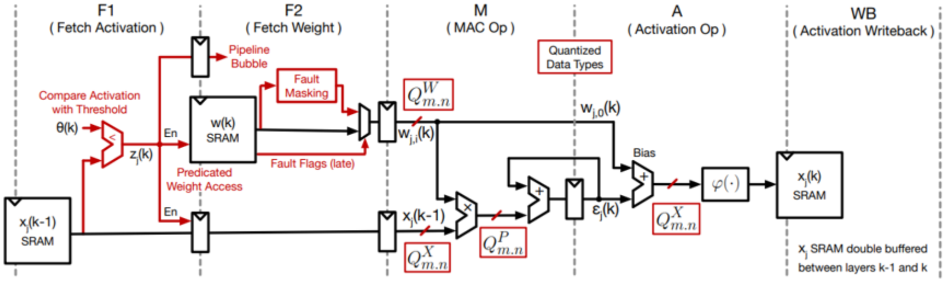


Figure 6.6. The microarchitecture of a single datapath lane. Modifications needed for optimizations are shown in red

Stage 3 of Minerva aggressively optimizes DNN bitwidths. The use of optimized data types is a key advantage that allows accelerators to achieve better computational efficiency than general-purpose programmable machines. In a DNN accelerator, weight reads and MAC operations account for most power consumption. Fine-grained per-type, per-layer optimizations significantly reduce power and resource demands. Figure shows three signals that we independently quantize: neuron activity, network weights and the product that determines the multiplier width. The notation $Q_{m.n}$ describes a fixed-point type of m integer and n fractional bits. Minerva considers all possible combinations and granularities of m and n for each signal within each network layer, independently.

The minimum number of bits required is set to be the point at which reducing the precision by 1 bit exceeds MNIST's error confidence interval of 0.14%. All data path types are set to the largest per-type requirement. While reducing layer wise produces power savings, it requires large number of unique SRAMs which increases area. Stage 4 of Minerva reduces the number of edges that must be processed in the dataflow graph. Using empirical analysis of neuron activity, we show that by eliminating operations involving small activity values, the number of weight fetch and MAC operations can be reduced without impacting accuracy. From figure we see that there are many zeros, skipping whose operations would save power from avoiding multiplications and SRAM accesses.

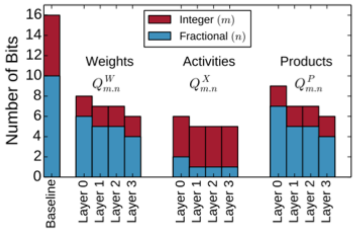


Figure 6.7. Minimum precision requirements for each datapath signal while preserving model accuracy within our established error bound

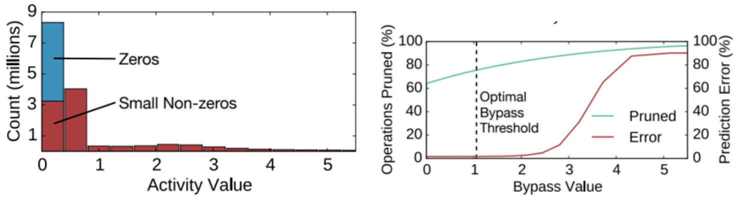


Figure 6.8. Analysis of neuron activities and sensitivity of prediction error to pruning. The vertical line corresponds to the point our error bound is exceeded.

Figure 6.7,6.8 shows that if we remove activities with magnitude less than 1.05, the overall prediction error is unaffected. This is because the rectifier output eliminates negative numbers producing zeros, which can be pruned to improve performance. In this way selective pruning reduces the power by 1.9X. The operations to be pruned cannot be determined directly. To achieve this, the data path lane splits the fetch operations over two stages. F1 reads the current neuron activation from SRAM and compares it with the per-layer threshold to generate a flag bit, which indicates if the operation can be skipped. Subsequently, F2 uses the flag to stall the following MAC using clock-gating to reduce the dynamic power of the data path lane. The hardware overhead for splitting the fetch operations, an additional pipeline stage and a comparator, are negligible.

The final stage of Minerva optimizes SRAM power by reducing the supply voltage. However, reducing voltage increases bit cell fault rate which is mitigated by co-designed fault mitigation techniques. Scaling SRAM voltages is challenging due to the low noise margin circuits used in SRAMs, including ratioed logic in the bit cell, domino logic in the bit line operation, and various self-timed circuits. Figure 6.9 shows the bit cell fault rates when scaling the supply voltage. So, Razor double sampling method is used for fault detection. Unlike parity, Razor monitors each column of the array individually, and hence there is no limit on the number of faults that can be detected, and information is available on which bit(s) are affected. The overheads for Razor SRAM fault detection are 12.8% and 0.3% for power and area.

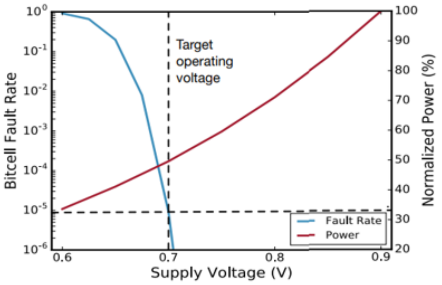


Figure 6.9. SRAM supply voltage scaling trends for fault rate and power dissipation

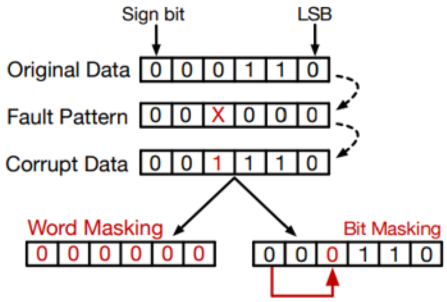


Figure 6.10. Illustration of word masking (faulty weights set to zero) and bit masking (faulty bits set to sign bit) fault mitigation techniques.

Razor SRAMs provide fault detection, not correction. A lightweight approach is used which does not reproduce the original data but instead attempts to mitigate the impact of intermittent bit-flips to the DNN's model accuracy. To prevent prediction accuracy degradation, we combine Razor fault detection with mechanisms to mask data towards zero when faults are detected at the circuit level. Masking can be performed at two different granularities: word masking: when a fault is detected, all the bits of the word are set to zero; and bit masking: any bits that experience faults are replaced with the sign bit. This achieves a similar effect to rounding the bit position towards zero. Figure 6.10 shows a simple illustration of word masking and bit masking fault mitigation. The combination of Razor fault detection and bit masking fault mitigation allows the weight SRAMs to tolerate 44 more faults than word masking. This drops power by 2.7X on an average. Bit masking requires modifications to the weight fetch stage (F2) of the data path lane.

6.1.4 Evaluation

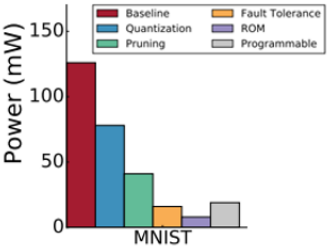


Figure 6.11. Results from applying the Minerva design flow to five application datasets to investigate general error.

Figure 6.11 shows the power reduction due to each step. On average, Minerva generated DNN accelerators dissipate 8.1 less power. There is tradeoff between specialization and power reduction. However, using programmable accelerator uses 2.4X more power. The largest overhead here is memory leakage.

6.1.5 Summary

An optimized hardware accelerator for deep neural networks that achieve minimal power consumption while maintaining high prediction accuracy is designed. Minerva is a holistic, highly automated co-design flow that combines insights and techniques across the algorithm, architecture, and circuit levels, enabling low-power accelerators for highly accurate DNN prediction. By aggressively optimizing data types, selectively pruning operations, and reducing SRAM voltages safely with novel fault mitigation techniques, Minerva can reduce the overall power consumption across five diverse ML datasets by an average of 8.1 without impacting prediction error. Minerva makes it possible to deploy DNNs as a solution in power-constrained mobile environments.

6.2 Templated Based Framework For Auto Deployment

6.2.1 Introduction

A wide range of commercial and enterprise applications rely on Machine Learning techniques. These have computationally intensive workloads which are repeated for large number of iterations. While the demand for these computationally intensive techniques increases, the benefits from general-purpose computing are diminishing. Programmable accelerators implemented on FPGAs, can provide large gains in efficiency and performance by restricting the workloads. The increasing availability makes them option to accelerate ML algorithms, however, development with FPGAs still need expertise in hardware design and implementation and the overall design cycle is long.

TABLA,[26] a template-based solution tackles this challenge by developing from circuit to programming model, for using FPGAs to accelerate statistical machine learning algorithms. It devises necessary programming abstractions and automated frameworks that are uniform across range of ML algorithms. It avoids higher level of abstraction from hardware design by leveraging commonalities in learning algorithms and expressing them as stochastic optimization problems. The learning models can be optimized using stochastic gradient descent where learning task becomes solving an optimization problem using SGD that iterates over the training data and minimizes an objective function. Although the SGD solver is mostly fixed across different learning algorithms, the objective function varies. Therefore, the accelerator for these learning tasks can be implemented as a template design, uniform across a set of machine learning algorithms which comprises of general framework. To be able to specialize the template design for a specific learning task, a hardware block implementing the gradient of the objective function for the algorithm needs to be designed and integrated. TABLA provides framework to automatically do the following. Therefore, with TABLA, the developer only needs to specify the learning model as the gradient of the objective function

TABLA automatically generates an accelerator for the specific learning algorithm while considering high-level design parameters of the target FPGA. TABLA-generated accelerators provide better speedups compared to CPU and GPUs. These benefits are achieved while the programmer write less than 50 lines of code. These results suggest that TABLA takes an effective step toward widespread use of FPGAs as an accelerator of choice for machine learning algorithms.

6.2.2 Overview

Machine learning generally involves two phases–the learning phase and the prediction phase. The learning phase generates a model that maps inputs onto outputs which then is used to predict the dependent variable for a new unseen input. The learning phase is more compute intensive and can benefit significantly from acceleration. Therefore, TABLA aims to provide solution that can automatically generate accelerators to accelerate the learning phase of a class of machine learning algorithms. Figure 6.12 illustrates an overview of TABLA workflow. It provides a high-level programming model that enables the programmers to specify the gradient of the objective function that captures the learning algorithm. TABLA focuses on learning algorithms as shown in figure 6.13 that can be implemented using stochastic gradient descent, therefore, the gradient of objective function is sufficient to generate the entire accelerator design. After providing the gradient of the objective function, one of the major components of TABLA, named the design builder, automatically generates the accelerator and its interfacing logic. The design builder uses a predefined set of accelerator templates to generate the accelerator.

The output of the design builder is a set of synthesizable Verilog codes. The inputs to the

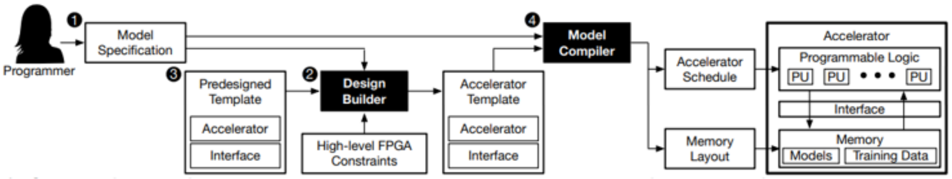


Figure 6.12. Overview of the workflow with Tabla.

design builder are the gradient function, high-level specification of the target FPGA hardware, a predesigned set of accelerator templates in Verilog. The predesigned templates used to design accelerator are generic and uniform across a large class of stochastic machine learning algorithms. These predefined templates are designed by expert hardware designers and comprise of both the accelerator and the interfacing logic that connects the accelerator to the rest of the system.

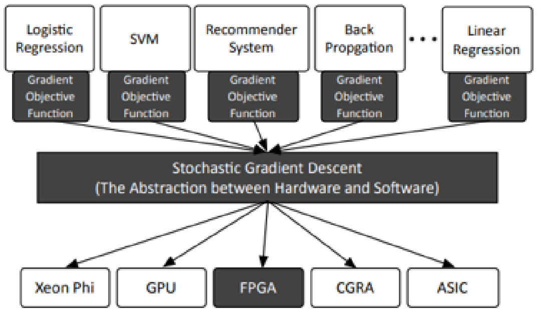

Logistic Regression
SVM
Recommender System
Back Propgation
Linear Regression
Gradient Objective Function
Gradient Objective Function
Gradient Objective Function
Gradient Objective Function
Gradient Objective Function
Stochastic Gradient Descent
(The Abstraction between Hardware and Software)
Xeon Phi
GPU
FPGA
CGRA
ASIC

Figure 6.13. Tabla leverages SGD as an abstraction between hardware and software to create a unified framework for accelerating ML algorithms.

Another component of TABLA is the model compiler that statically generates an execution schedule for the accelerator. The inputs to the model compiler are the structure of the accelerator and the specification of the gradient function. This converts the gradient function to a dataflow graph and augments it with the dataflow graph of the gradient descent. Then, it uses a minimum-latency resource-constrained scheduling algorithm to generate the accelerator schedule. It generates an order for the model parameters that will be learned, and this will determine the layout of parameters in the memory and streamlines the interfacing logic that communicates with the memory. As Figure depicts, TABLA can potentially target different platforms and new backends need to be developed for each target.

SGD forms the abstraction between hardware and software for TABLA that generates machine learning accelerators and therefore forms the template-base of TABLA. SGD is an optimization algorithm that aims to find the set of parameters that minimize a function. Each machine learning task in our target class is characterized by its objective function which learns a set of parameters. The objective function quantifies the error between the predicted value of the output and the actual output value corresponding to an input dataset. The ML algorithm learns the model by solving an optimization problem that minimizes the objective function over the entire training data according to:

$$min_{w^t \varepsilon R} \sum_i f\left(w_i^t x_i\right) \tag{6.1}$$

In the equation 6.1, w^t represents the parameters of the model that needs to be minimized, x_i is the input and $f\left(w_i^t x_i\right)$ is the objective function which is minimized using SGD optimization algorithms. SGD starts with an initial set of parameter values and iteratively minimizes the function. This iterative minimization is achieved by taking steps in the decreasing direction of the function's derivative or gradient which is:

$$w^{t+1} = w^t - \mu \times \frac{\partial\left(\sum_i f\left(w_i^t x_i\right)\right)}{\partial w^t} \tag{6.2}$$

As the above equation 6.2 shows, w^{t+1} goes in the negative direction of $\partial f / \partial w$ with a rate μ. That is, in a single iteration of gradient descent, it calculates the derivative of the objective function over the entire training data and generates the next set of parameters $\left(w^{t+1}\right)$ as shown by equation. The overhead of large training data is avoided using SGD where it divides the objective function into smaller differentiable functions requiring a single element. Therefore, the gradient of the smaller function is calculated only over a single element. The equation for stochastic gradient descent transforms into:

$$w^{t+1} = w^t - \mu \times \frac{\partial f(w_i^t x_i)}{\partial w^t} \tag{6.3}$$

SGD takes more iterations to converge to minimum value, however, the benefits obtained by avoiding the data access to all the input elements for each iteration is higher than the cost incurred by having more iterations.

6.2.3 Architecture

The programmer needs to express different learning algorithms by specifying the gradient of the objective function as shown in algorithm 2. Programming interface is a high-level interface that enables the representation of ML algorithms close to the mathematical models, it comprises of language constructs and keywords that are commonly seen in several statistical ML algorithms. The programming interface comprises of two types of constructs, data declaration and mathematical operations. Data declarations enable the programmer to specify the different data elements which include model input, model output, model parameters, gradient, and iterators. The model_input keyword refers to a single input dataset while the model_output declaration refers to the corresponding output provided as the training data. Both these data types are inputs to the machine learning task and are read-only while the ML algorithm learns the model. The model keyword refers to the model parameters that get updated every iteration. The iterator declaration enables the programmer to declare the dimensions of arrays. Moreover, iterators also clearly depict the autonomy of operations.

Mathematical operations allow the programmer to express different operations and functions which are subdivided into three categories like basic, group and nonlinear. The basic operations constitute mathematical operations like $-, +, <, >, *$ and require two arguments A and B. Group Operations are performed over a group of elements and includes the following operations,$\sum$ (sum),$\prod$ (group multiply), and $\|\|$ (norm). Also, these operation types require an iterator argument to operate on a group of elements. They produce an output with dimension one less than the input dimension. Nonlinear Operations constitute functions like Log, Sigmoid, Gaussian, and Sigmoid Symmetric. The output has the same dimensionality as the input as this operation is performed element by element. Using the data and operation language declarations defined , programmer can represent several statistical ML algorithms.

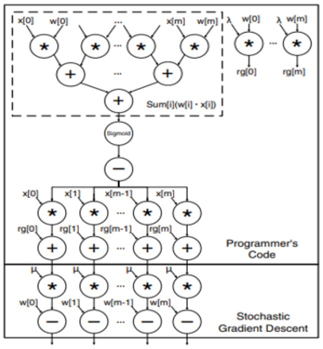


Figure 6.14. A complete dataflow graph of the logistic regression algorithm

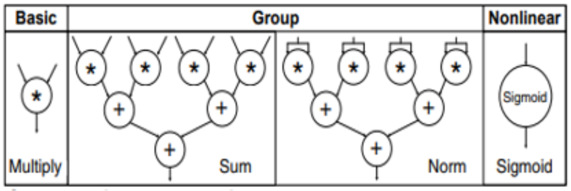


Figure 6.15. Dataflow graph for basic, group and nonlinear type of operations

Algorithm 2. Laguage declaration

```
**model_input** $x[m]$;  //model input features
**model_output** $y'[n]$;  //model outputs
**model** $w[n][m]$;  //model parameters
**gradient** $g[n][m]$;  //gradient

**iterator** $i[0:m]$;  //iterator for group operations
**iterator** $j[0:n]$;  //iterator for group operations

//m parallel multiplications followed by
//an addition tree; repeat n times in parallel
$s[j] = sum[i](x[i] * w[j][i])$;

$y[j] = sigmoid(s[j])$;  //n parallel sigmoid operations
$e[j] = y[j] - y'[j]$;  //n parallel subtractions
$g[j][i] = x[i] * e[j]$;  //n*m parallel multiplications
$rg[j][i] = \lambda * w[i][j]$;  // n*m parallel multiplications
$g[j][i] = g[j][i] + rg[j][i]$;  // n*m parallel additions
```

The above code 2 shows how the gradient of logistic regression can be expressed in a few lines using TABLA's programming interface. In this code the programmer first declares the data types: model_input, model_output, model, and the gradient. Then, two iterators i and j are declared as the model values are two dimensional. Next, operations are performed over the declared data types beginning with the sum operation. This operation performs multiplication x[i] * w[j][i] and adds up all the multiplication results into a single result (s[j]) in the i dimension assuming a constant j. Finally, the result generated by this code is the gradient for the given model_input, model_output and model. Several ML algorithms can be represented using the above-mentioned language declarations.

After the programmer provides the gradient of the objective function, TABLA's model compiler first integrates this objective function with the stochastic gradient descent. To generate a concrete accelerator, the model compiler then generates a dataflow graph as shown in figure 6.14,6.15 that can be mapped and scheduled on hardware. Dataflow graphs are intermediate representations that can be translated into the accelerator and its execution schedule. Thus, the final phase of compilation is the scheduling phase in which the compiler generates a static schedule for the learning task that is represented by a dataflow graph.

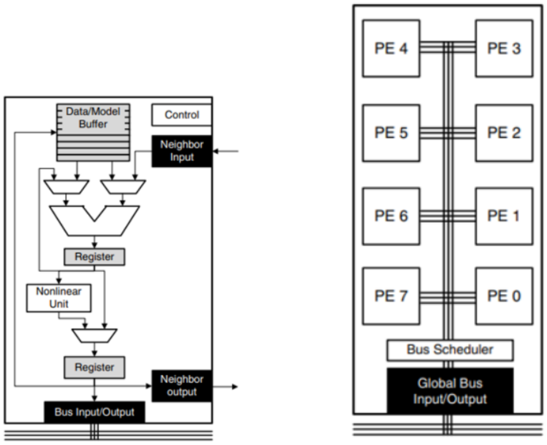
The model compiler appends the above generate DFG with SGD. The Figure shows the data-flow graph for at least one operation of each type - basic, group and nonlinear. These dataflow graphs show the input and output edges along with the intermediate nodes that perform the computation of each operation. The group operations involve more than one computational node. The dataflow graph also depicts the opportunities for parallelism that will be exploited by the hardware accelerator. The model compiler combines the DFG of individual operations according to the code that expresses the gradient. The DFG for a ML task can be generated by combining the DFG of each operation with the code of the gradient function as shown in figure. After the compiler framework generates the DFG, different scheduling algorithms can be used to schedule each operation in the DFG. We perform this scheduling using a Minimum Latency Resource Constrained Scheduling algorithm, which schedules operations given a limited set of resources. The above graph is scheduled using As Soon As Possible algorithm in which operations are scheduled as soon as all the predecessors of an operation are completed. This algorithm is faster but consumes lot of area, hence Minimum Latency Resource Constrained

Scheduling is used. This algorithm assumes that operations are single step and use a single type of resource which is valid as the base design comprises of processing engines which just take one step to generate results.

Distance from sink of an operation is the number of operations that need to be performed after the op to reach the final output/sink. This quantifies the priority of each operation where higher the distance from sink, the higher its priority is. So, in scheduling algorithm, an operation is scheduled at cycle if all the predecessors have been scheduled and completed, it has the highest priority among the unscheduled ready ops and resource is available to accommodate the op. The algorithm terminates when all the operations are successfully scheduled. After the schedule for operations is generated, TABLA framework generates the design for hardware accelerator that can accommodate this schedule.

Owing to flexibility and reconfigurability FPGAs are used to accelerate machine learning tasks. TABLA's compilation framework produces different schedule for different learning algorithms. Thus, we propose a reconfigurable accelerator design that can accommodate these schedules and can be specialized to accelerate a range of ML algorithmsas shown in figure 6.17. The fundamental and reconfigurable component of this accelerator architecture is a Processing Engine as shown in figure 6.16. The components within a PE and its interconnection with other PEs are customized and designed to accelerate the learning algorithm.

As the figure illustrates, PEs comprises of a computational unit that performs calculations and a storage unit that stores the model parameters and data elements. PEs include some fixed components like the ALU, Data/Model Buffer, Register and Bus while others can be reconfigured. As all the statistical ML tasks have some form of mathematical operation making the ALU a crucial component of PE. A buffer is necessary to store the model or other incoming data from external DRAM. The register is used for some group operations such as sum ($\sum$) or pi ($\prod$). Finally, a bus interface is crucial and is reserved for retrieving training data and model parameters from the external memory. However, the communication with other PEs is not always required and is dependent on the algorithm. The exchangeable components in a PE include specialized



(a)) Processing Engine (PE)

(b) Processing Unit (PU)

Figure 6.16. (a) A PE comprising of compute and memory units.(b) PU comprising of 8 processing engines connected through a intra-PU bus.

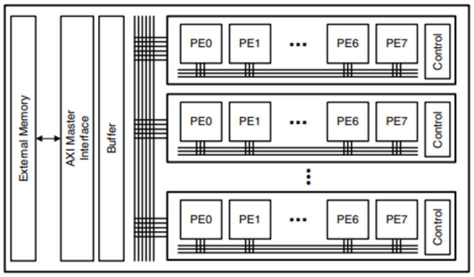


Figure 6.17. Accelerator design showing the processing units and processing engines.

control unit, nonlinear unit, multiplexers, and the neighbor input and output communication. In the above example it is only used for summation output, hence using a nonlinear unit in every PE wastes area and power. Therefore, a nonlinear unit is only provided in the final PE which accrues results. Finally, communication between neighboring PUs is useful for algorithms that combine data. Reading the neighbor's result will avoid contention on the bus if multiple PEs need data from the other PEs. Allowing neighboring PE communication leverages spatial locality. Once the PE is finalized in congruence with algorithm, it is incorporated into the processing unit.

Processing Unit contains eight identical PEs as shown. Although the design can scale to larger or smaller numbers of PEs, frequency is maximum with 8 PEs. The bus between the PEs is referred to as the intra-PU bus and between the PUs is referred to as inter-PU bus. As shown the design comprises of multiple PUs connected through a pipelined global bus. This pipelined global bus also connects our design to the AXI interface which in turn connects to an external memory. The training input and output data is transferred from the external memory to the programmable logic after every iteration of the learning algorithm. The initial model parameters are only transferred once at the start of the execution. The number of PUs in the design are dependent on the algorithm being implemented and varies accordingly.

6.2.4 Evaluation

Tabla framework is evaluated by implementing hardware on FPGA platform. It outperforms ARM by an average speed up of 15.0. While Maximum speedup of 46X has been obtained for Reco model due to relatively larger model topology which provides greater opportunity for parallelism. However, GPUs outperform Tabla, Tesla provides 59X and GTX provides 15.5X in comparison to Tabla which provides 15X speedup. As the speedup results show, the TABLA-generated FPGA accelerators provide significant speedup over both multicore CPUs and the Tegra K1 GPU within a limited power budget of 2W. We compare the performance-per-watt to understand the benefits of FPGA acceleration without the variations in the power budget. TABLA, on average, achieves 30.1 and 81.7 over ARM and Xeon, respectively. On the GPU side, TABLA's FPGA accelerators provide 22.7×, 53.7*x*, and 30.6*X* higher performance-per-Watt compare to Tegra, GTX 650, and Tesla GPUs as shown in 6.7,6.8.

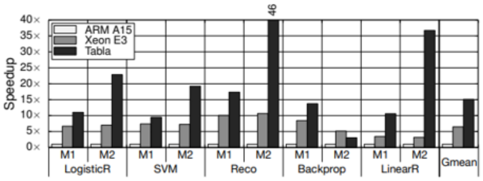


Figure 6.18. Speedup of Tabla in comparison to a range of CPU and GPU platforms.

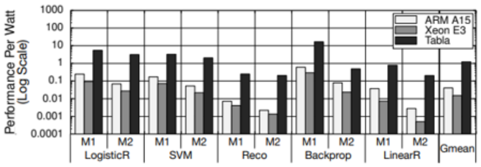


Figure 6.19. Comparison of performance-per-Watt between CPUs, GPUs and Tabla.

The TABLA-generated FPGA accelerators close the performance gap but provide much higher efficiency and operate at lower power budget. Area depends on the model size. Backpropagation utilizes least area because of its smaller model size while Reco uses more area as its default configuration is larger. Empirically, a PU design with 8 PEs strikes a balance between frequency and intra PU parallelism. The number of PE and PU can be reconfigured according to the algorithm. As shown in figure 6.20,6.21 increasing number of PEs leads to linear increase in speedup. However, beyond a certain number of PEs we either observe diminishing returns or a decrease in the speedup as the parallelism in the algorithm is limited and increasing would lead to wastage of resources. The frequency change is negligible as frequency increase with

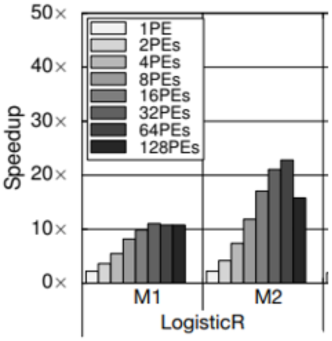


Figure 6.20. Speedup change for varying number of PEs in the design in comparison to ARM CPU.

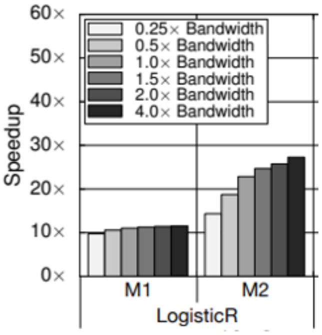


Figure 6.21. Speedup with varying Bandwidth for Tabla generated accelerator in comparison to ARM CPU.

increasing PEs is countered by requirement of wider global bus. Data transferred from external memory to the accelerator uses the AXI interface which has limited bandwidth. The bandwidth can be bottleneck at very low values such as 0.25 of the default bandwidth. As the bandwidth increases the speedup starts to increase but observe diminishing returns beyond a point. By providing a bandwidth that is 4 the default value the speedup numbers only increase by 60% of the default speedup.

6.2.5 Summary

ML algorithms include compute-intensive workloads that can benefit significantly from acceleration. FPGAs are an attractive platform for accelerating these important applications. However, FPGA design still requires relatively long design cycles and extensive expertise in hardware design. TABLA bridges the gap between the machine learning algorithms and the FPGA accelerators. It leverages SGD as the abstraction between hardware and software to automatically generate accelerators for a class of statistical machine learning algorithms. Compared to CPU,GPU the TABLA generated accelerators deliver an average speed up of 2.9 and 30.6 higher performance-per-Watt, respectively. These gains are achieved while the programmers only write less than 50 lines of code. These results suggest that TABLA takes an effective step in a widespread use of FPGAs for machine learning algorithms.

6.3 Conclusion

Both the design flow explained above, one optimizes for flexibility and other optimizes for power have been presented. The integration of which can be left as a future work. While there is a large body of research on the design of deep learning accelerators, only a few of them have focused on adopting them in data centers. We believe it is necessary to revisit server architectures to support accelerators. In addition, we need to consider both power and flexibility in order to accommodate multiple machine learning algorithms apart from just deep neural networks. We also need to address OS level challenges such as sharing accelerators among different processes and scheduling tasks on multiple processes on an accelerator. We believe that exploring these lines of research will unearth significant benefits from investing in automated design development and mixed-signal devices, accelerators.

Chapter 7

Conclusion

As we have seen now a days deep neural networks are used everywhere. It is a corner stone of modern AI systems. Its enables application ranging from smart assistance, image/speech recognition to self-driving cars, modern health care. But these new AI applications brings new challenges to the underlying hardware systems and infrastructure. Example we need high performance throughput for applications like self-driving cars which deal with a large amount of data. We also need low latency in smart assistance to have a smooth conversation. In addition, energy efficiency is becoming important now a days as many devices run on battery. With the end of Dennard scaling and diminishing benefits from transistor scaling general purpose architectures could no longer support the challenges as we have discussed. This led to rise of domain specific architectures which perform specific functions more efficiently in contrast to general purpose CPUs.

Accelerator design has come a long way. In related work we have see the performance improvements that are made. Starting with design of main compute block, there are many improvements done like data flow modelling, NOC architecture design, sparse networks and so on. Also new methods like analog computing, real time AI, tensor processing units, multi chip accelerators have been discussed. Most of these accelerators have made their way into consumer electronics. But their limited computational capacity still necessitates offloading most of the inference task to cloud. InFaaS has become the backbone of deployed applications like voice assistants smart speakers, and enterprise applications. As the demand for INFaaS scales, one solution could be continuously increasing the number of accelerators in the cloud. Although intuitive, this approach is neither cost-effective nor scalable with the ever-increasing demand for DNN services. On the other hand, multi-tenancy, where a single node is shared across multiple requests, has been a primary enabler for the success of cloud-computing in current scale. Without multi-tenancy, it is hard to even fathom the progress and future of datacenters and cloud-based computing. Nonetheless, multi-tenancy has not been a primary factor in the design of DNN accelerators because of the arms race to design the fastest accelerator, the utmost recency of accelerator adoption in datacenters, and challenges associated with multi-tenancy in accelerators. Planaria explore this timely, yet unexplored dimension of multi-tenancy in the architecture design of DNN accelerators. The key idea is dynamically fissioning the DNN accelerator at runtime to spatially co-locate multiple DNN inferences on the same hardware.

This idea is relatively new and has lot of scope for improvements which are discussed as part of future work. Even though we have dynamic allocation of multiple DNN inferences the power consumption remains same. This could be enhanced by using automated flows discussed to improve the power requirements. Also current design is restricted to only DNN which can be extended to many machine learning algorithms by automating the design flow to improve both power, flexibility and throughput.

In conclusion, hardware accelerator design has come a long way and there is still scope for lot of improvements. The main idea of neural networks started by imitating the brain functionality. Going in the same lines trying to imitate the nature helps us design high computing engines with low power consumption.

Bibliography

[1] Vahideh Akhlaghi, Amir Yazdanbakhsh, Kambiz Samadi, Rajesh K Gupta, and Hadi Esmaeilzadeh. Snapea: Predictive early activation for reducing computation in deep convolutional neural networks. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 662–673. IEEE, 2018.

[2] Jorge Albericio, Patrick Judd, Tayler Hetherington, Tor Aamodt, Natalie Enright Jerger, and Andreas Moshovos. Cnvlutin: Ineffectual-neuron-free deep neural network computing. *ACM SIGARCH Computer Architecture News*, 44(3):1–13, 2016.

[3] Guoyang Chen and Xipeng Shen. Free launch: optimizing gpu dynamic kernel launches through thread reuse. In *Proceedings of the 48th International Symposium on Microarchitecture*, pages 407–419, 2015.

[4] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. *ACM SIGARCH Computer Architecture News*, 42(1):269–284, 2014.

[5] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, and Temam Olivier. Dadiannao: A machine-learning supercomputer. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 609–622. IEEE, 2014.

[6] P. Chi, S. Li, C. Xu, T. Zhang, J. Zhao, Y. Liu, Y. Wang, and Y. Xie. Prime: A novel processing-in-memory architecture for neural network computation in reram-based main memory. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 27–39, 2016.

[7] Sudipto Das, Vivek R Narasayya, Feng Li, and Manoj Syamala. Cpu sharing techniques for performance isolation in multi-tenant relational database-as-a-service. *Proceedings of the VLDB Endowment*, 7(1):37–48, 2013.

[8] Christina Delimitrou, Daniel Sanchez, and Christos Kozyrakis. Tarcil: reconciling scheduling speed and quality in large shared clusters. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, pages 97–110, 2015.

[9] Hadi Esmaeilzadeh, Emily Blem, Renee St Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *2011 38th Annual international symposium on computer architecture (ISCA)*, pages 365–376. IEEE, 2011.

[10] J. Fowers, K. Ovtcharov, M. Papamichael, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, L. Adams, M. Ghandi, S. Heil, P. Patel, A. Sapek, G. Weisz, L. Woods, S. Lanka, S. K. Reinhardt, A. M. Caulfield, E. S. Chung, and D. Burger. A configurable cloud-scale dnn processor for real-time ai. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 1–14, 2018.

[11] Udit Gupta, Samuel Hsia, Vikram Saraph, Xiaodong Wang, Brandon Reagen, Gu-Yeon Wei, Hsien-Hsin S Lee, David Brooks, and Carole-Jean Wu. Deeprecsys: A system for optimizing end-to-end at-scale neural recommendation inference. *arXiv preprint arXiv:2001.02772*, 2020.

[12] Rehan Hameed, Wajahat Qadeer, Megan Wachs, Omid Azizi, Alex Solomatnikov, Benjamin C Lee, Stephen Richardson, Christos Kozyrakis, and Mark Horowitz. Understanding sources of inefficiency in general-purpose chips. In *Proceedings of the 37th annual international symposium on Computer architecture*, pages 37–47, 2010.

[13] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A Horowitz, and William J Dally. Eie: efficient inference engine on compressed deep neural network. *ACM SIGARCH Computer Architecture News*, 44(3):243–254, 2016.

[14] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A Horowitz, and William J Dally. Eie: efficient inference engine on compressed deep neural network. *ACM SIGARCH Computer Architecture News*, 44(3):243–254, 2016.

[15] Nikos Hardavellas, Michael Ferdman, Babak Falsafi, and Anastasia Ailamaki. Toward dark silicon in servers. *IEEE Micro*, 31(4):6–15, 2011.

[16] Vighnesh Iyer Alon Amid Howard Mao John Wright Colin Schmidt Jerry Zhao Albert Ou Max Banister Yakun Sophia Shao Borivoje Nikolic Ion Stoica Krste Asanovic Hasan Genc, Ameer Haj-Ali. Gemmini: An agile systolic array generator enabling systematic evaluations of deep-learning architectures. *arXiv preprint arXiv:1911.09925*, 2019.

[17] Kartik Hegde, Jiyong Yu, Rohit Agrawal, Mengjia Yan, Michael Pellauer, and Christopher Fletcher. Ucnn: Exploiting computational reuse in deep neural networks via weight repetition. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 674–687. IEEE, 2018.

[18] Norman P. Jouppi, Cliff Young, Nishant Patil, David A. Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, Richard C. Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark

Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-datacenter performance analysis of a tensor processing unit. *CoRR*, abs/1704.04760, 2017.

[19] Patrick Judd, Jorge Albericio, Tayler Hetherington, Tor M Aamodt, and Andreas Moshovos. Stripes: Bit-serial deep neural network computing. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–12. IEEE, 2016.

[20] Harshad Kasture and Daniel Sanchez. Ubik: efficient cache sharing with strict qos for latency-critical workloads. *ACM SIGPLAN Notices*, 49(4):729–742, 2014.

[21] Duckhwan Kim, Jaeha Kung, Sek Chai, Sudhakar Yalamanchili, and Saibal Mukhopadhyay. Neurocube: A programmable digital neuromorphic architecture with high-density 3d memory. *ACM SIGARCH Computer Architecture News*, 44(3):380–392, 2016.

[22] Sheng Li, Jung Ho Ahn, Richard D Strong, Jay B Brockman, Dean M Tullsen, and Norman P Jouppi. The mcpat framework for multicore and manycore architectures: Simultaneously modeling power, area, and timing. *ACM Transactions on Architecture and Code Optimization (TACO)*, 10(1):1–29, 2013.

[23] Sheng Li, Ke Chen, Jung Ho Ahn, Jay B Brockman, and Norman P Jouppi. Cacti-p: Architecture-level modeling for sram-based structures with advanced leakage reduction techniques. In *2011 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 694–701. IEEE, 2011.

[24] Robert LiKamWa, Yunhui Hou, Julian Gao, Mia Polansky, and Lin Zhong. Redeye: analog convnet image sensor architecture for continuous mobile vision. *ACM SIGARCH Computer Architecture News*, 44(3):255–266, 2016.

[25] Shaoli Liu, Zidong Du, Jinhua Tao, Dong Han, Tao Luo, Yuan Xie, Yunji Chen, and Tianshi Chen. Cambricon: An instruction set architecture for neural networks. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 393–405. IEEE, 2016.

[26] Divya Mahajan, Jongse Park, Emmanuel Amaro, Hardik Sharma, Amir Yazdanbakhsh, Joon Kyung Kim, and Hadi Esmaeilzadeh. Tabla: A unified template-based framework for accelerating statistical machine learning. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 14–26. IEEE, 2016.

[27] Pascale Minet, Eric Renault, Ines Khoufi, and Selma Boumerdassi. Analyzing traces from a google data center. In *2018 14th International Wireless Communications & Mobile Computing Conference (IWCMC)*, pages 1167–1172. IEEE, 2018.

[28] Nishant Patil David Patterson Gaurav Agrawal Raminder Bajwa Sarah Bates Suresh Bhatia Nan Boden Al Borchers Rick Boyle Pierre-luc Cantin Clifford Chao Chris Clark Jeremy

Coriell Mike Daley Matt Dau Jeffrey Dean Ben Gelb Tara Vazir Ghaemmaghami Rajendra Gottipati William Gulland Robert Hagmann C. Richard Ho Doug Hogberg John Hu Robert Hundt Dan Hurt Julian Ibarz Aaron Jaffey Alek Jaworski Alexander Kaplan Harshit Khaitan Daniel Killebrew Andy Koch Naveen Kumar Steve Lacy James Laudon James Law Diemthu Le Chris Leary Zhuyuan Liu Kyle Lucke Alan Lundin Gordon MacKean Adriana Maggiore Maire Mahony Kieran Miller Rahul Nagarajan Ravi Narayanaswami Ray Ni Kathy Nix Thomas Norrie Mark Omernick Narayana Penukonda Andy Phelps Jonathan Ross Matt Ross Amir Salek Emad Samadiani Chris Severn Gregory Sizikov Matthew Snelham Jed Souter Dan Steinberg Andy Swing Mercedes Tan Gregory Thorson Bo Tian Horia Toma Erick Tuttle Vijay Vasudevan Richard Walter Walter Wang Eric Wilcox Norman P. Jouppi, Cliff Young and Doe Hyun Yoon. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pages 1–12, 2017.

[29] Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Brucek Khailany, Joel Emer, Stephen W Keckler, and William J Dally. Scnn: An accelerator for compressed-sparse convolutional neural networks. *ACM SIGARCH Computer Architecture News*, 45(2):27–40, 2017.

[30] Brandon Reagen, Paul Whatmough, Robert Adolf, Saketh Rama, Hyunkwang Lee, Sae Kyu Lee, José Miguel Hernández-Lobato, Gu-Yeon Wei, and David Brooks. Minerva: Enabling low-power, highly-accurate deep neural network accelerators. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 267–278. IEEE, 2016.

[31] Hardik Sharma, Jongse Park, Emmanuel Amaro, Bradley Thwaites, Praneetha Kotha, Anmol Gupta, Joon Kyung Kim, Asit Mishra, and Hadi Esmaeilzadeh. Dnnweaver: From high-level deep network models to fpga acceleration. In *the Workshop on Cognitive Architectures*, 2016.

[32] Hardik Sharma, Jongse Park, Naveen Suda, Liangzhen Lai, Benson Chau, Vikas Chandra, and Hadi Esmaeilzadeh. Bit fusion: Bit-level dynamically composable architecture for accelerating deep neural network. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 764–775. IEEE, 2018.

[33] Lalith Suresh, Peter Bodik, Ishai Menache, Marco Canini, and Florin Ciucu. Distributed resource management across process boundaries. In *Proceedings of the 2017 Symposium on Cloud Computing*, pages 611–623, 2017.

[34] Abdulaziz Tabbakh, Murali Annavaram, and Xuehai Qian. Power efficient sharing-aware gpu data management. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 698–707. IEEE, 2017.

[35] Balajee Vamanan, Hamza Bin Sohail, Jahangir Hasan, and TN Vijaykumar. Timetrader: Exploiting latency tail to save datacenter energy for online search. In *Proceedings of the 48th international symposium on microarchitecture*, pages 585–597, 2015.

[36] Nandita Vijaykumar, Kevin Hsieh, Gennady Pekhimenko, Samira Khan, Ashish Shrestha, Saugata Ghose, Adwait Jog, Phillip B Gibbons, and Onur Mutlu. Zorua: A holistic approach to resource virtualization in gpus. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–14. IEEE, 2016.

[37] Yuzhao Wang, Lele Li, You Wu, Junqing Yu, Zhibin Yu, and Xuehai Qian. Tpshare: a time-space sharing scheduling abstraction for shared cloud via vertical labels. In *Proceedings of the 46th International Symposium on Computer Architecture*, pages 499–512, 2019.

[38] Bo Wu, Guoyang Chen, Dong Li, Xipeng Shen, and Jeffrey Vetter. Enabling and exploiting flexible task assignment on gpu through sm-centric program transformations. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, pages 119–130, 2015.

[39] Zhifeng Chen Quoc V. Le Mohammad Norouzi Wolfgang Macherey Maxim Krikun Yuan Cao Qin Gao Klaus Macherey Jeff Klingner Apurva Shah Melvin Johnson Xiaobing Liu Łukasz Kaiser Stephan Gouws Yoshikiyo Kato Taku Kudo Hideto Kazawa Keith Stevens George Kurian Nishant Patil Wei Wang Cliff Young Jason Smith Jason Riesa Alex Rudnick Oriol Vinyals Greg Corrado Macduff Hughes Jeffrey Dean Yonghui Wu, Mike Schuster. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*, 2016.

[40] Shijin Zhang, Zidong Du, Lei Zhang, Huiying Lan, Shaoli Liu, Ling Li, Qi Guo, Tianshi Chen, and Yunji Chen. Cambricon-x: An accelerator for sparse neural networks. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–12. IEEE, 2016.

[41] Yuhao Zhu, Matthew Halpern, and Vijay Janapa Reddi. Event-based scheduling for energy-efficient qos (eqos) in mobile web applications. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 137–149. IEEE, 2015.

[42] Yuhao Zhu, Daniel Richins, Matthew Halpern, and Vijay Janapa Reddi. Microarchitectural implications of event-driven server-side web applications. In *Proceedings of the 48th International Symposium on Microarchitecture*, pages 762–774, 2015.