January 2016

ACCURACY AND PERFORMANCE IMPROVEMENTS IN CUSTOM CNN ARCHITECTURES

Aliasger Zaidy
Purdue University

Graduate School Form 30
(Revised 08/14)

PURDUE UNIVERSITY
GRADUATE SCHOOL
Thesis/Dissertation Acceptance

This is to certify that the thesis/dissertation prepared

By Aliasger Zaidy

Entitled: Accuracy and Performance Improvements in Custom CNN Architectures

For the degree of Master of Science in Electrical and Computer Engineering

Is approved by the final examining committee:

EUGENIO CULURCIELLO

ANAND RAGHUNATHAN

VIJAY RAGHUNATHAN

To the best of my knowledge and as understood by the student in the Thesis/Dissertation Agreement, Publication Delay, and Certification/Disclaimer (Graduate School Form 32), this thesis/dissertation adheres to the provisions of Purdue University's "Policy on Integrity in Research" and the use of copyrighted material.

EUGENIO CULURCIELLO

Approved by Major Professor(s): _____

Approved by: V. Balakrishnan 07/25/2016

Head of the Department Graduate Program Date

ACCURACY AND PERFORMANCE IMPROVEMENTS IN CUSTOM CNN ARCHITECTURES

A Dissertation

Submitted to the Faculty

of

Purdue University

by

Aliasger T. Zaidy

In Partial Fulfillment of the

Requirements for the Degree

of

Master of Science in Electrical and Computer Engineering

August 2016

Purdue University

West Lafayette, Indiana

ACKNOWLEDGMENTS

I thank Dr. Eugenio Culurciello for the inspiring and insightful discussions without which this text would not have been possible. I am grateful to him for giving me this opportunity to work under his supervision.

I would also like to thank Dr. Vijay Raghunathan and Dr. Anand Raghunathan for their comments and continued guidance.

I would like to thank my parents who have been a constant source of moral support and encouragement. I am grateful to Vinayak Gokhale for our spirited discussions and debate sessions which were the source of new ideas. I have learned a great deal while working with him and picked up several skills which I will carry forward in my career.

I would also like to thank other members at e-Lab: Abhishek Chaurasia, Andre Chang, Alfredo Canziani, SangPil Kim, Jonghoon Jin and Aysegul Dundar. They have helped me in various ways throughout my Masters. Last but not the least, I would like to thank my friends Rishi Naidu, Jhalak Patel and Yunus Akhtar who made my Masters at Purdue enjoyable.

TABLE OF CONTENTS

Page

LIST OF TABLES

LIST OF FIGURES

ABBREVIATIONS

ASIC	Application Specific Integrated Circuit
AXI	Advanced eXtensible Interface
CNN	Convolutional Neural Networks
CPU	Central Processing Unit
DNN	Deep Neural Networks
FP	Floating Point
FPGA	Field Programmable Gate Array
GPU	Graphics Processing Unit
MAC	Multiply ACcumulate
NN	Neural Network
SRAM	Static Random Access Memory

ABSTRACT

Zaidy, Aliasger T. M.S.E.C.E., Purdue University, August 2016. Accuracy and Performance Improvements in custom CNN Architectures. Major Professor: Eugenio Culurciello.

Convolutional Neural Networks (CNNs) are biologically inspired feed forward artificial neural networks. The artificial neurons in CNNs are connected in a manner similar to the neurons in the mammalian visual system. CNNs are currently used for image recognition, semantic segmentation, natural language processing, playing video games and many other applications. A CNN can consist of millions of neurons that require billions of computations to produce a single output.

Currently CNN workloads are accelerated by GPUs. While fast, GPUs are power hungry and are not feasible in mobile and embedded applications like car, home automation, etc. Recently interest has surged in developing FPGA/ASIC based novel architectures for processing using CNNs in real time while keeping the power budget low. The current generation of custom architectures utilize either single or half precision floating point or 16bit Q8.8 fixed point processing elements.

However, floating point hardware is larger and slower than fixed point hardware, especially in FPGAs, which have dedicated fixed point units but no floating point units. Due to this, choosing a number format becomes a performance vs accuracy tradeoff. In this work, we aim to test various number representation schemes and their effect on the bandwidth requirements and accuracy of a custom CNN architecture. We also test architectural changes that improve the throughput of the processing elements. Together, we expect to improve both accuracy and performance of a custom CNN accelerator architecture. The system is prototyped on the Xilinx Zynq Series Devices.

1. INTRODUCTION

Over the past few years embedded vision systems utilizing convolutional neural networks [1–5] have grown extensively. These networks have a wide variety of applications in fields like robotics, self driving vehicles, home automation, etc. Apart from object classification and detection from images, nowadays these networks are also capable of pixel wise classification/semantic segmentation, object localization, natural language processing, video processing and playing games.

Due to their popularity and state of the art performance, convolutional neural networks have been widely accepted in the industry with companies like Google and Microsoft contributing to their proliferation via networks like Inception [6], GoogLeNet [7] and Resnet [8]. Bing image searches are being performed using CNNs, self driving cars use semantic segmentation CNNs for parsing video feed [9–11], CNNs are being fitted into intruder detection systems. There are many more such examples.

One of the major bottlenecks in improving the performance of CNNs has been computational power using conventional resources such as CPUs and GPUs. CPUs are control based and fail to efficiently handle the computationally intensive CNN workload. The recent advancement in GPU architecture and process technology make them suitable for CNNs but they end up consuming a lot of power (around 300W). GPUs also require a batch of data (say 16/32 images) in order to optimally utilize their computational capability. While power and data requirement for optimal utilization are not important in a research setting, they become a major consideration when designing real life systems such as mobile chips, automotive controllers and server chips. Power consumption requirement in servers might seem counter-intuitive but one must remember that reducing power consumption in servers has an exponential benefit since it reduces the cost and power required to cool the servers also.

Due to these reasons, FPGAs are finding increased use in accelerating CNNs [12–15]. FPGAs are flexible and hence can be reprogrammed with a new architecture unlike GPUs which need to be replaced (incurring a high cost). FPGAs also consume much less power compared to GPUs. It is evident that the flexibility comes with the cost of being less computationally powerful than GPUs. However, number of Giga operations per second per watt are far larger for an FPGA compared to a GPU.

Given these advantages of FPGAs, a number of novel architectures utilizing FPGAs for running CNN workloads have been developed such as Neuflow [16], nn-X [17], Catapult [18] and Snowflake to name a few. Neuflow was one of the first neuromorphic chips developed by Fabaret et al. nn-X is the successor of Neuflow that overcomes a lot of the bandwidth limitations present in the Neuflow convolver units. Catapult is a system developed by Microsoft Research using Altera FPGAs and is currently employed in running Bing image searches and for scientific research at the Texas Advanced Computing Center in Austin. Snowflake is an architecture developed by Gokhale et al at Purdue University that achieves state of the art performance for CNNs. In this thesis, we analyze and propose improvements with reference to the Snowflake architecture that would enable it to perform at a better accuracy and greatly increase it computational capability. The improvements aim at utilizing FPGA resources efficiently and harnessing resources left currently unused by Snowflake. We experiment with various numeric representation techniques and optimizations to achieve a final accuracy of 0.84% compared to 6.92% obtained with conventional snowflake. The total power consumption on the Zedboard does not increase significantly and the DSP unit utilization increases from 192 to 204.

1.1 Convolutional Neural Networks

A CNN consists of several convolutional feature extractor layers followed by a multiclass classifier. Figure 1.1 shows a typical object detection/classification network architecture which consists of convolution, maxpooling (downsampling), ReLU (non

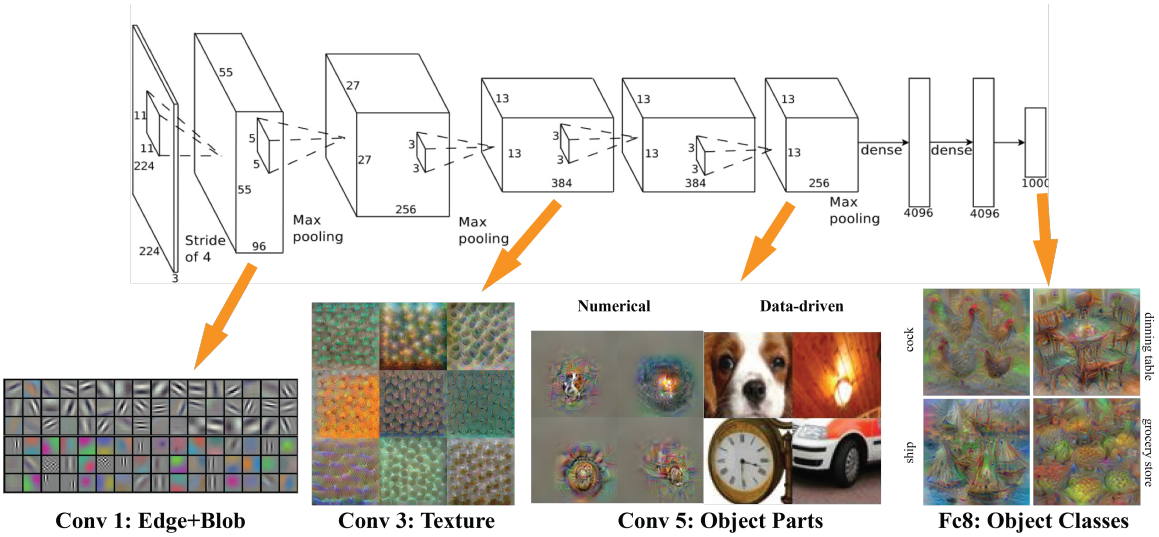


Fig. 1.1.: A typical convolutional neural networks for object detection consisting of convolutional layers followed by a multi class classifier for generic multi class object recognition. The network works on arbitrarily large images, once trained, to produce a classification map as the output. Adapted from `https://github.com/donglaiw/mNeuron/blob/master/V_neuron_single.m` by D Wei, 2016, Retrieved from URL. Copyright (c) 2015 donglai

linear) and fully connected layers while a typical encoder-decoder semantic segmentation network consists of all of the above except the fully connected layers. Semantic segmentation also introduces a new operation of deconvolutions which are used to reverse the effects of convolutions. Mathematically deconvolutions are convolutions with inverted weights.

The *convolution layers* are basically filters for extracting features from the output of the previous layers. The *maxpool layers* are used to summarize the output of the convolution layers and reduce overfitting while *ReLU* introduces nonlinearity in the network. In this work, these three layers are together referred to as a *layer*. The inputs of layer are connected to the output of the previous layer except for the first layer whose input is a signal usually an image. The input and outputs of a layer

are called feature maps. The *fully connected layers* are classifiers that use the feature maps from the final convolution layer to find the objects present in the original image.

A convolution is an operation in mathematics which when performed on two functions produces a third function that expresses the amount of overlap of one function over another. Mathematically,

$$h[m,n] = \sum_{j=1}^{kH} \sum_{i=1}^{kW} f[m+i, n+j] * g[i,j] \tag{1.1}$$

In CNNs a convolution is used as a filter for extracting features from the input [19]. Inputs are signals such as audio, video, images, etc. This work focuses mainly on image datasets.

The non-linearity separates the input into distinct linearly independent outputs. It usually models the firing rate of a biological neuron. A linear function may be used in its place but then the output would be a set of linearly dependent layers that could be coupled into a single huge layer. A linear activation function would increase the computation but not necessarily make the network more expressive [20]. Popular non linear functions are the sigmoid, the hyperbolic tan and the rectified linear unit (ReLU).

Pooling reduces translational invariance and thus avoids overfitting. Hence, if a pixel moves in the pooling region, the output would not change. Pooling also reduces the number of computations encountered in the later layers. Usually the number of maps increases as we go deeper and pooling helps reduce the width and height to compensate for the increase in the depth. These days 1x1 convolutions are being used to reduce the computational complexity as well [7].

1.1.1 CNN information flow

Let us consider the CNN described in Figure 1.1. Here the network consists of a total of 8 computation stages viz. 5 convolutional layers and 3 fully connected layers.

ReLU is applied to the output of every stage. Max pooling is applied to the output of the first two stages.

The input to the first stage is a RGB image which is convolved with a kernel of size 11x11. The convolution is done with a stride of 4 which means that after each computation the kernel window is shifted by 4 units to find the next output pixel. If we need N output feature maps we use N kernels each of size kWxkHx3 where kW and kH are 11 in this case. Here N is taken as 96 giving us 96 output feature maps of size 55x55. These output feature maps are then maxpooled using a 2x2 maxpool with stride 2 giving 96 maps of size 27x27.

This is done 4 more times in order to extract finer features. At the end of the five convolutional layers, we have 256 maps with size 13x13. These maps are then used as input to the first fully connected layer. The fully connected layers are like traditional multi layered perceptrons (MLPs). The output of the first fully connected layer is fed to another one before being passed to a 1000-way softmax in order to obtain the distribution over a 1000 class labels.

1.2 Outline

This section details the organization of this document. The next chapter goes through the history of hardware accelerators for convolutional neural networks and existing research in reducing power/bandwidth of custom hardware while keeping the accuracy constant. It talks about popular present day architectures. The next section also describes the existing methodologies for reducing precision in CNNs. In Chapter 3, we describe the hardware details of suggested improvements and the rationale behind them. Chapter 3 also presents the variable precision fixed point algorithm and details how the proposed strategy fits into an existing architecture. Chapter 4 details the experimental setup and results. It also compares the results to existing numeric representations. Chapter 5 analyzes the accuracy and performance improvements and hypothesizes the reason for sub-optimal performance of popular representations.

Chapter 6 concludes the discussion by summarizing the work described and providing suggestions for interesting future extensions for this project.

2. LITERATURE REVIEW

Convolutional neural networks were proposed a long time ago but were widely accepted for computer vision only after their use in object detection and classification was shown by Krizhevsky at al. [1]. They have become popular recently due to the increase in computational power and the advent of fast, high throughput GPUs systems. Networks like Inceptionv3 [6] and ResNet-34 [8] require a massive 5.71 G-ops and 3.6 G-ops respectively. They are usually trained and tested on high-end GPU clusters or supercomputers like the NVIDIA DGX-1.

While feasible for training a network, clusters and supercomputers are unsuitable for day to day scenarios such as self-driving car or home automation systems. These systems are to be operated within a power budget and using a 300W GPU is an extremely critical design consideration. While some home consumers might be willing to run a 300W GPU, we must also consider that home security systems need to run on battery in case of power outage and a GPU will be unusable in such circumstances.

In order to account for these situations several ASIC and FPGA based custom architectures have been proposed in recent times. Low power consumption is inherent for most FPGA and ASIC based architectures. These architectures mainly consider scalability, programmability and i/o/memory bandwidth.

Scalability encompasses computation capability and power consumption of the system. Scalability is a design consideration because all devices may not necessarily run the same CNN. Home automation devices may not need the same network architecture or computation power as a performance and accuracy critical application such as a self-driving car. On the other hand a 50 W device in a single home security camera is a big design choice while being a small unnoticeable dent in a cars power budget.

Programmability is one of the key considerations while developing a custom CNN architecture. While a custom architecture might have 100% efficiency in order for it to be popular, it must be friendly to software developers. A software developer does not need to be taxed with the micro architectural details. A C/Python interface which allows the user to make API calls in order to process their neural networks is critical.

I/O bandwidth is important because in a practical system, the data is fetched via pins and sent out to an output device such as a LED display. While we would like to store all the data on the chip itself, it is currently unfeasible to do so. Hence, most of the data is stored in off chip DRAM and needs to be continuously swapped in and out to compute a single output feature map. If the memory bandwidth is low, training and testing networks will take a long time.

We will be using the snowflake architecture for the purpose of this work although the concepts are general and can be applied to any CNN coprocessor. Snowflake is currently implemented using FPGAs. It is a SIMD architecture and has its own instruction set. It functions as a coprocessor relying on the host to initiate the first instruction which DMAs a program to on chip memory from DRAM. The coprocessor then takes over and starts computing the network based on the program written.

The smallest unit in Snowflake is a compute unit which is a 256 bit vector multiplier accumulator with 16 bit granularity. Each compute unit has its own kernel cache and shares a unified image cache with three other compute units. Four compute units together make up a compute cluster. Four such compute clusters comprise one snowflake. Each snowflake also has a scalar pipeline and scalar and vector register files. A snowflake also has a unified L2 cache that is shared by the four compute clusters.

Recently there has been a lot of interest in reducing the size of the network since it will allow for lower bandwidth, possibly faster computation and lower power consumption. There are ways like dropout and drop connect that are used to eliminate some pixels probabilistically in training. However, such networks when tested at 32

bit floating point precision consume a lot of bandwidth and power. Several methodologies have been proposed for reducing the size of the network. There is biological motivation too as [21] notes that the neurons in our brain have 6 to 12 bit precision.

One solution by Courbariaux et al. stochastically binarizes the weights. The gradients are not binarized during back propagation but are instead passed in 32 bit floating point format. The parameter update is done in full precision because stochastic gradient descent, the most popularly used back-propagation method, uses infinitesimally small changes in the parameters in order to adjust the final output as desired. If the weights were binarized during update, the system may even fail to converge. However, this work has been tested on small CNNs such as those required for MNIST and SVHN and it achieves an accuracy drop of up to 10% for these small networks.

Another method proposed by [22] is to use 8 bit floating point precision. Their experiments are however limited to 10%-20% of values present in the network. It can be shown that for a network 10% of the weights may lie in a small region of the weights-value distribution and hence, the 1% accuracy drop claimed may not hold true when the entire network is considered.

[23] has evaluated the use of Q4.12 and Q2.14 schemes for training and test purposes. The units are hardcoded into a systolic array of MACc units on a Kintex FPGA. Their experiments on the CIFAR10 and MNIST datasets yield a 1% drop in accuracy.

Deep compression [24, 25] is a methodology proposed by Han et al. that quantizes the weights and then performs Huffman encoding in order to reduce their sizes further. This methodology proposes the following flow: prune connections by learning the important ones, reduce precision of the weights depending on their distribution and then perform Huffman encoding to reduce the size of the network further. While this methodology improves the average time per layer by 4x for a batch size of 1, it also degrades the average time per layer by 5x for a batch size of 64. This is because of

inefficient utilization of CPU/GPU resources while doing sparse matrix computations on a large batch of images.

Our aim is to provide a methodology that provides consistent improvement in Gops per sec per watt and bandwidth across batch sizes for networks like Alexnet [1]. We also aim to analyze the statistical distribution of values in each layer and dynamically provide an optimum representation for each layer.

3. HARDWARE/REPRESENTATIONS OVERVIEW

In the previous two chapters, we reviewed the various architectures and learning representations that have been proposed to process the computationally intensive CNNs. We also stated that our aim was to optimize the G-ops/s-Watt and memory bandwidth while being agnostic to batch size and network architecture and keeping the accuracy as close as possible to that obtained with single precision.

3.1 Statistical distribution of weights

Before going into the representation and learning details, let us look at the distribution of the weights in a trained CNN, as a whole and on a per layer basis. We will be using the network shown in Figure 1.1 as a reference in this document and if any other network is referenced, it will be explicitly mentioned.

Figure 3.1 shows that the parameters in a network are distributed along a Normal distribution and this is true whether we are considering the entire network or each layer individually. Here the weights are tightly packed around a central value and there are very few outliers. These outliers can be characterized by the ceiling and the floor values of the representation.

Based on this information, we can deduce that while using a typical number representation such as single precision or half precision floating point approximately half of the useful range is thrown away. The loss is even greater for Q8.8 fixed point representation where we are using only the first 256 values out of 65536 possible combinations which is an effective utilization of 0.4% of the entire range. In a custom CNN accelerator, using these representations will result in a significant portion of the bandwidth being used just to transmit zeros. This gives us the motivation to search for a suitable representation for the parameters and pixels in a CNN. We would

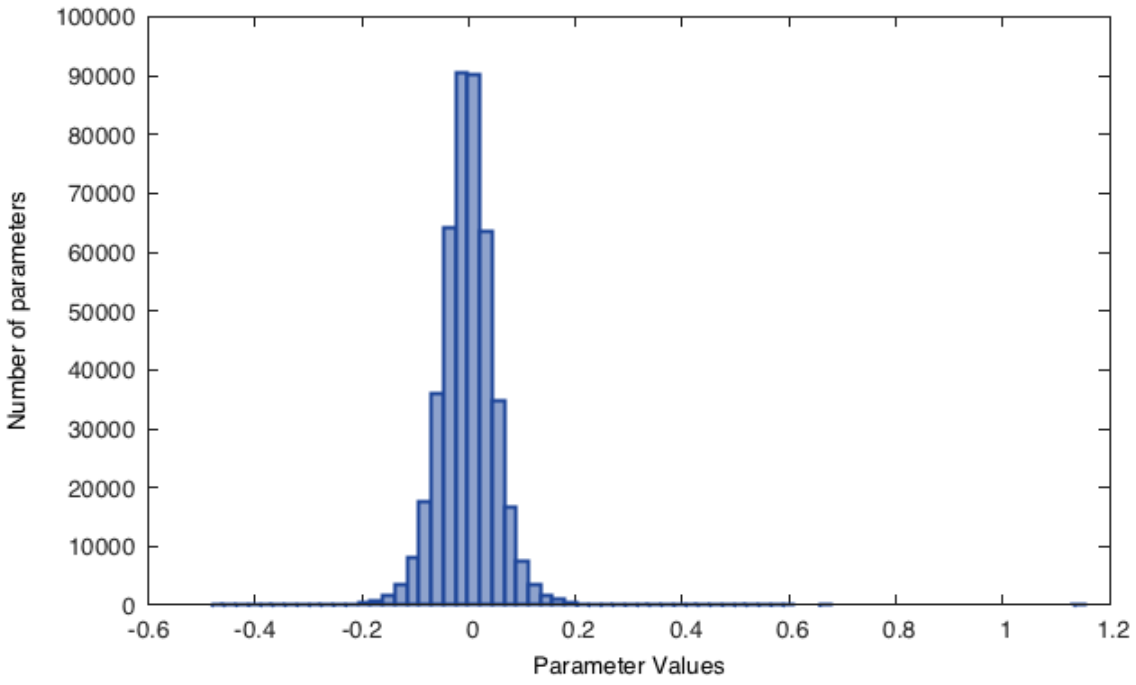


Fig. 3.1.: Distribution of Weights and Biases in a typical CNN. From the graph we can see that the parameters of a CNN lie approximately on a Normal Distribution. Due to this if we use a typical number representation scheme like single precision, most of the representations are unused.

typically like this representation to be applicable to all existing hardware resources and to not require retraining of CNNs.

3.2 Possible Representations

3.2.1 Floating Point

Floating point representation usually consists of 3 parts: the sign bit, the mantissa and the exponent. Consider a number [s,e,m] where s, e and m denote the sign bit, exponent and mantissa respectively. The numerical value of this number is $s * (1.m) * (2^{(e-E)})$ where E is the bias which is usually $\max(e) + 1/2$.

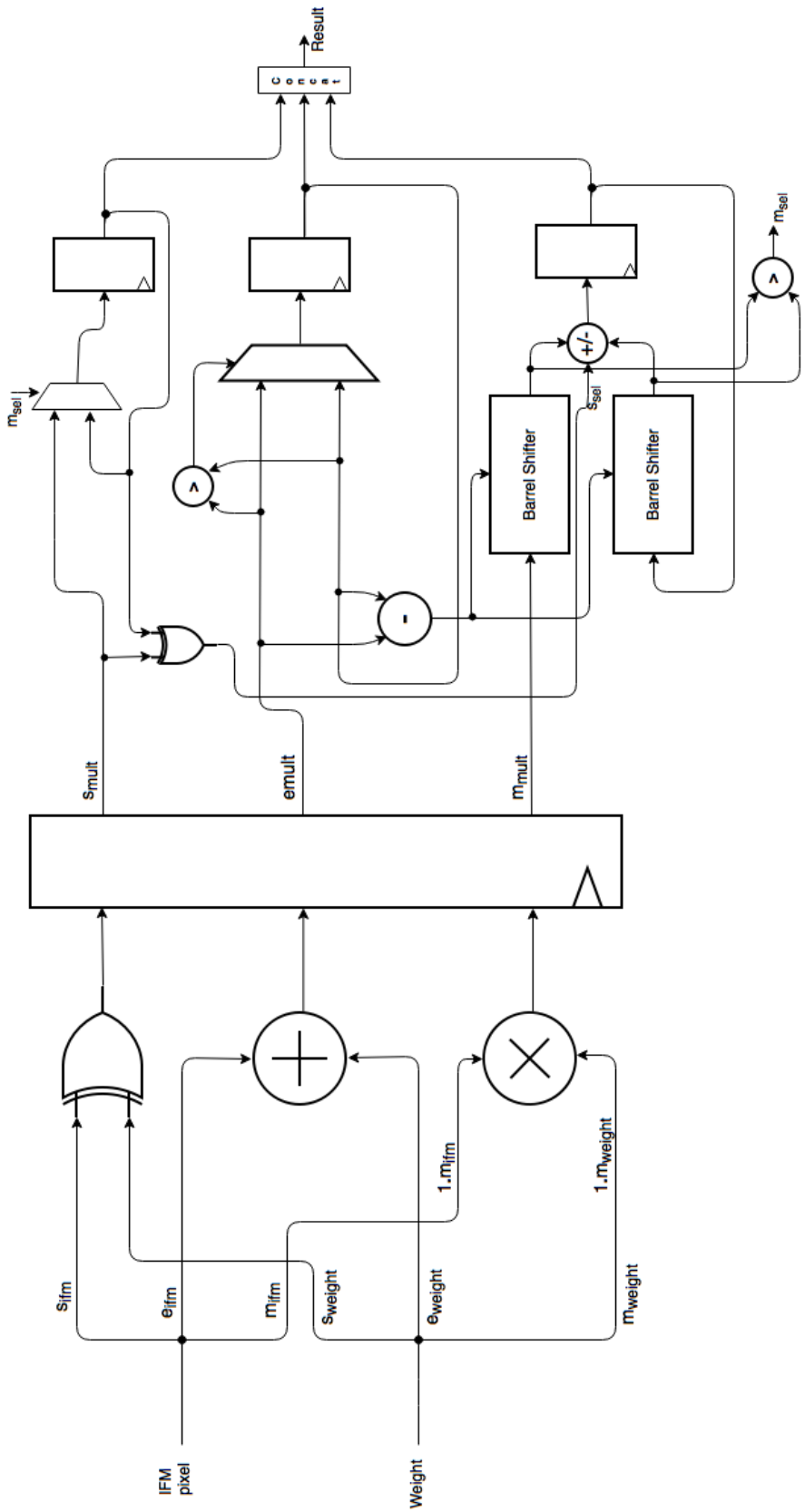


Fig. 3.2.: Floating Point Multiply Accumulator

Note than the mantissa has m+1 bits with the leading bit being either 0 or 1. When the exponent e is non zero the leading bit is 1 i.e. the normal range and when it zero the leading bit is zero i.e. the subnormal range.

A typical floating point multiply accumulator unit is shown in the Figure 3.2. Let us consider the multiplication module of the FP MAC. When multiplying two numbers the result is positive if both the numbers have the same sign and negative if they are of opposite signs. A XOR gate is used to deduce the sign bit of the multiplication result.

In order to understand the operations on the exponent and mantissa let us consider two numbers $m_1 * 2^{e_1}$ and $m_2 * 2^{e_2}$. When we multiply these two numbers on paper,we add the exponents and multiply the mantissas to get $m_1 * m_2 * 2^{e_1+e_2}$. This is achieved using the adder and the multiplier modules shown in the Figure 3.2. The outputs of the XOR, multiplier and adder modules are registered and passed to the accumulator stage.

The accumulator is a floating point adder with its output connected to one of its inputs. Let us consider $s_1 * m_1 * 2^{e_1}$ and $s_2 * m_2 * 2^{e_2}$ as the inputs to the floating point adder. The exponents are first compared and the greater of the two is passed on to the result. We use the difference between the greater and the smaller exponent in order to shift the mantissa of the number with the greater exponent to the right using a barrel shifter. The shifted mantissa is added to the mantissa of the larger number in order to obtain the final mantissa for the result. These two mantissas are also compared and their result is used to determine the sign of the output.

After the last value is accumulated a done signal, not shown in the Figure, puts the accumulated value on the output bus and clears the accumulator register in the next cycle. The result of the accumulation is then passed on to the normalization unit which eliminates some precision and packs the number back into the original format to be used for the next layer. In a conventional floating point unit if the result is greater/lesser than the maximum/minimum possible numerical values for

the representation, a Nan value is returned. In our implementation, we return the maximum/minimum possible numerical value or a zero respectively.

16 bit floating point/half precision

Fig. 3.3.: Floating Point Representation

Let us start with the most popular representation of all, the 16 bit floating point. In recent times this representation has gained a lot of momentum even causing NVIDIA to rethink their GPU architecture by introducing vector 16bit floating point multiply accumulators. It has been well established that half precision arithmetic gives accuracy close to single precision for CNNs. Figure 3.3 gives the distribution of bits in half precision. Typically, there are 10 bits reserved for the mantissa, 5 for the exponent and 1 bit for the sign. We however have implemented a flavor which has 11 bits for the mantissa and only 4 for the exponent. The reason for doing this is evident from the statistical distributions. The values of weights and inputs never exceed $\pm$ 32 and hence using 4 bits of exponent provides us an additional bit of precision for the mantissa.

12 bit floating point

Another possible representation uses 12 bits: 7 for the mantissa, 4 for the exponent and 1 for the sign. From the structure itself, it is evident that this representation will have lesser accuracy than the 16 bit floating point but the overall effect of the drop in precision on the network accuracy is less. The maximum achievable range is approximately the same as 16 bit floating point. Since the mantissa multiplier and

accumulator in this representation are 8 bits wide, this representation utilizes the FPGA fabric instead of being synthesized as a DSP unit.

8 bit floating point

8 bit floating point representation is also known as minifloat. A minifloat is showing great promise for NNs these days. It comprises of 3 bits for mantissa, 4 for the exponent and the remaining bit for sign. We have also experimented with 4 mantissa and 3 exponent bits. Here the precision and range are greatly reduced and while it would work for small NNs and datasets like LeNet/MNIST, in order to get it to work for modern NNs significant tweaks in the network would be necessary.

3.2.2 Fixed Point

Fixed point representation uses a fixed decimal point. A fixed point number has the same precision across the entire range as opposed to a floating point number who precision decreases monotonically as the numerical value increases. A fixed point number is usually represented as $QX.Y$ where X denotes the number of integer bits while Y denotes the number of bits in the fractional part.

16 bit fixed point $Q8.8$

In the Q88 format the integer part is allocated 8 bits and the fractional part gets 8 bits. This allows the maximum possible value to be 127.99609375 and the precision is 0.00390625. This may seem as a good representation at first but the error may accumulate at each layer and cause significant difference in the output compared to baseline.

Dynamic fixed point

This is a variation of the fixed point format in which the decimal point is adjusted in order to achieve the best possible accuracy. This format has been used for experimenting with 16, 12 and 8 bit numbers. Most commonly used formats are 16bit wide Q4.12 and Q5.11. In our experiments, we consider the distribution of weights and select the format that would allow maximum coverage.

While experimenting we observed that different layers work better at different precisions. Due to this we decided to alter the precision of the weights on a per layer basis. Algorithm 1 is used for this purpose.

Algorithm 1 CNN Inference using Dynamic Fixed Point Computation. Here the quantize routine converts between precision formats. The findQuantizationParameters routine finds the conversion ratio for the net layer and passes it to the quantize routine.

Require: weights W, biases B, input feature map ifm, network model net, initial representation QX.Y

Ensure: output feature map ofm

1. **Input Quantization**

   For w in W to end, w = quantize(w)

   For b in B, b = quantize(b)

   For $pixel$ in ifm, $pixel$ = quantize($pixel$)

2. **Compute Output Feature Map**

   $ofm = net$:forward(ifm)

3. **Output Module**

   $quantparams$ = findQuantizationParameters(ofm)

This algorithm works well in software as well as hardware. In terms of the CNN accelerators, two additional modules are included, one at the input and one at the output of the design. The result from each compute element in hardware is truncated to the desired precision format. Since the format of the inputs is known we know exactly where the output decimal point lies. The output unit uses this information and the required output format information to truncate number to the desired precision. The required output format information depends on the distribution of the inputs to the system. The input unit is responsible for computing this distribution and providing the necessary information to the output unit.

3.3 Multi clock design

Multiple clock domains are a common trick used in modern day CPUs and GPUs to operate different parts of a design at appropriate clock rates. Clock domain crossing allows a SoC designer to interface components operating asynchronously with the CPU to the base system and to each other.

Modern FPGAs are equipped with PLLs that can generate in-phase/out-of-phase clocks upto GHz frequencies. Unfortunately, operating a big design at high frequencies on FPGAs requires a great amount of effort to achieve timing closure. Hard IP like DSP units and on chip RAMs on FPGAs can be operated at frequencies as high as 600 MHz even when the fabric clocks at 150-250 MHz.

In this work, we experiment with CDC techniques to operate the DSPs faster than the rest of the design. We have a DMA fetching Data into BRAMs on a Xilinx FPGA. The BRAMs are instantiated inside an asynchronous FIFO module (shown in Figure 3.4) and output data at 1.5x, 2x and 3x the fabric frequency. The DSP units clocked at higher frequencies use this data output from BRAMs to generate the output feature maps.

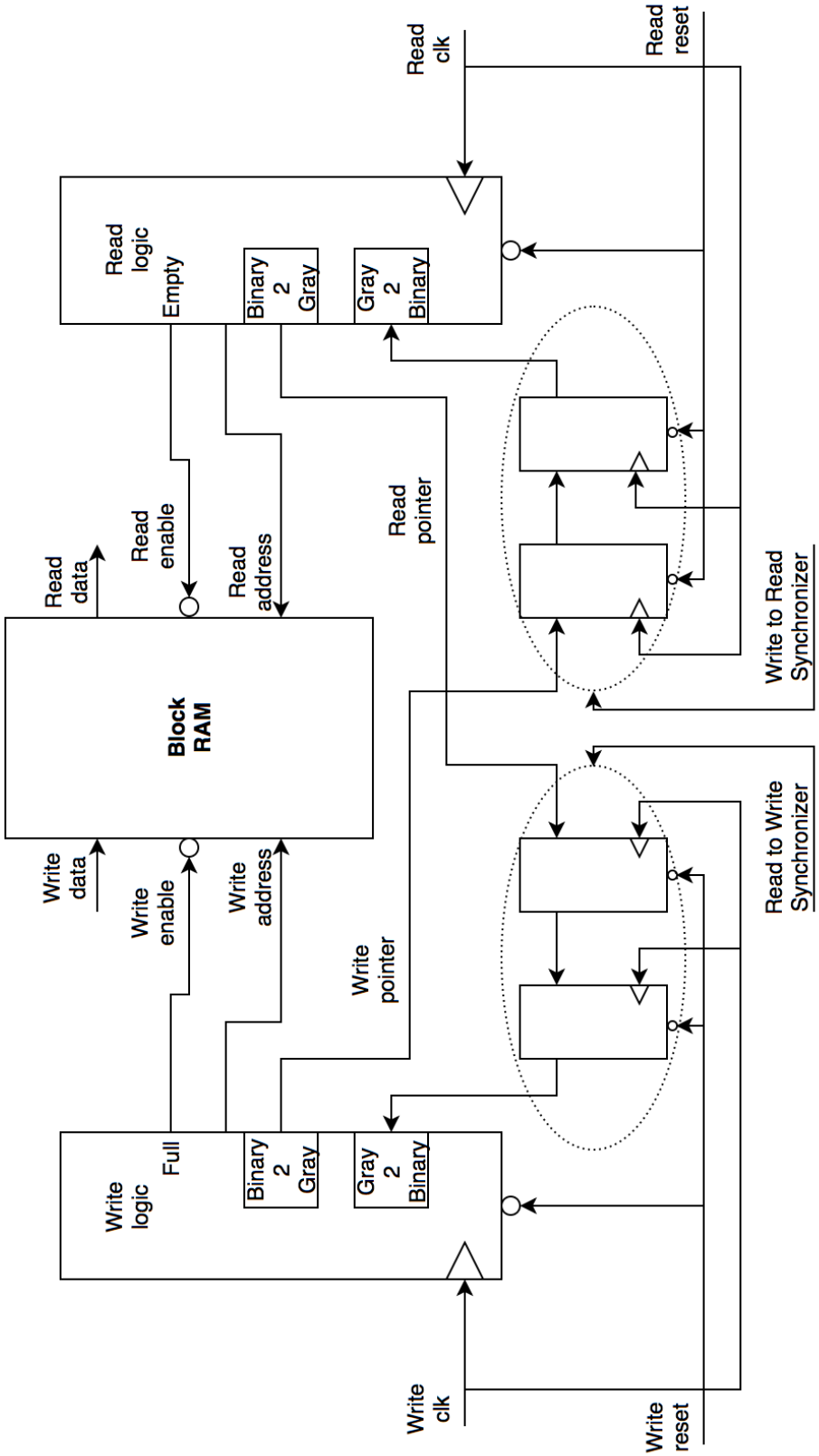


Fig. 3.4.: Asynchronous FIFO

This technique is useful as long as we keep the DSP units fed with data. If we are unable to achieve the desired bandwidth, the compute elements will be underutilized and the increase in power due to multiple frequency domains will not be compensated by a proportionate increase in throughput.

The Asynchronous FIFO shown in Figure 3.4 consists of three important parts: the FIFO pointer logic, the synchronizers and the RAM block. Here the pointer logic is divided into read and write modules where the former handles the empty signal and the pop pointer while the latter handles the full signal and the push pointer. These modules also have binary to gray and gray to binary logic. The binary to gray logic converts the push/pop pointers from binary to gray format in order to be sent to the other module. This is done because synchronizers can correctly handle only single bit changes across clock domains and gray code always changes by a single bit when the pointer is incremented. The gray to binary module converts the incoming pop/push pointer to binary so that it can be compared to the internal push/pop pointer in order to assert the full/empty signal respectively.

The synchronizers are just a pair of FFs that convert a pointer from one clock domain to another. The RAM block is the memory module that stores the data. The pointer logic is pessimistic since the full/empty signals are not deasserted for two cycles after a pop/push has occurred. This compute unit is placed between two such asynchronous FIFOs.

3.4 System Overview

The snowflake architecture shown in Figure 3.5 is used as a reference design. This work is mainly concerned with the parts highlighted in red is shown in Figure 3.5. We work towards finding the best possible implementation for the compute units using the DSP48E1s on the Zynq7000 series. The dynamic fixed point input unit sits on the AXI stream [26] interface coming into the snowflake co-processor. The corresponding output unit is placed between the compute units and the vector registers. When

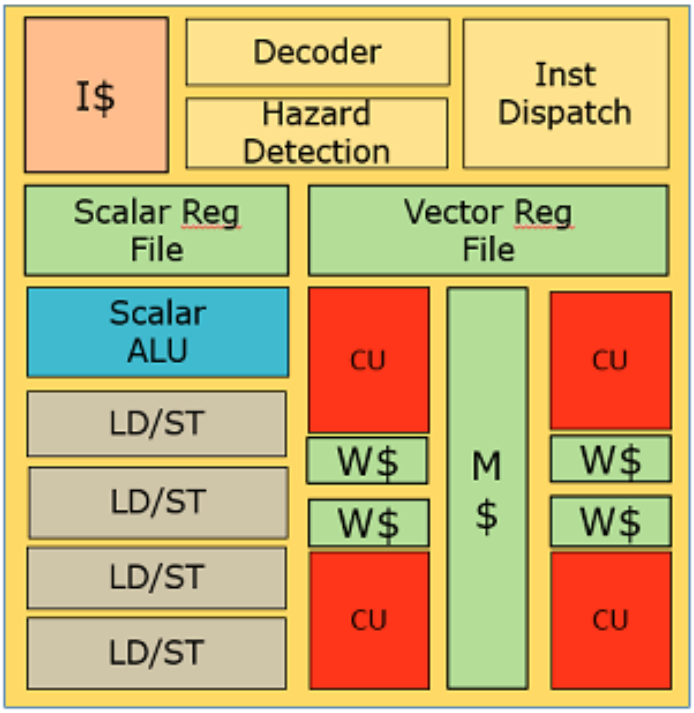


Fig. 3.5.: Snowflake Architecture by Gokhale et al.

going for multiple clock domains, small asynchronous FIFOs are placed at the output of the M$ and W$ and after the output unit.

4. RESULTS

Let us now take a look at the performance results obtained using the methodologies described in the previous chapters.

4.1 Experimental Setup

The improvements suggested in this work were tested on the Zynq Zedboard development platform [27] (see Table 4.1). The Zedboard has the Zynq 7z020 chip which consists of two parts: the Processor System (PS) and the Programmable Logic (PL). The PS has two ARM Cortex-A9 cores and is connected to 512MB DDR3 memory. The PL can fit upto 12 compute units each with 16 DSPs. The PL was clocked at 187.5 MHz.

Table 4.1: Zynq Zedboard Hardware Specifications

Platform	Zynq Zedboard
Chip	Xilinx Zynq 7z020 SoC
Processor	Dual ARM Cortex-A9 @ 667 MHz
Programmable Logic	Artix-7
Memory	512 MB DDR3 @ 533 MHz
Memory Bandwidth	4.2 GB/s full duplex
PL Frequency	187.5 MHz

Torch7 [28] and Thnets[1] were the main software tools used in the work. Torch7 is a scientific computing framework built on the Lua programming language. Thents is a set of C libraries by e-lab that opens and parses a trained Torch7 model.

4.2 Accuracy

Table 4.2: Percentage Error per layer for the reference network shown in Figure 1.1 across different precision formats. The error is with respect to the single precision floating point format. Here layers l1-l5 consist of convolution and ReLU. Layers l2 and l3 have maxpooling. Layers l6-l8 are linear layers

	l1	**l2**	**l3**	**l4**	**l5**	**l6**	**l7**	**l8**
16bit FP	0.04	0.01	0.003	0.05	0.13	0.42	0.23	0.39
12bit FP	0.08	0.37	0.41	0.31	0.65	0.93	2.46	1.27
8bit FP	3.35	4.21	4.97	5.13	8.24	9.67	10.06	15.03
Q8.8 fixed	0.95	1.65	2.30	2.84	2.80	9.13	10.49	6.92
Q7.9 fixed	0.61	0.86	1.21	1.51	1.48	4.60	5.23	3.18
Q6.10 fixed	0.27	0.42	0.58	0.71	0.71	2.23	2.61	1.67
Q5.11 fixed	0.13	0.22	0.31	0.37	0.37	1.17	1.31	0.82
Q4.12 fixed	0.07	1.26	3.36	5.50	7.42	17.8	15.5	10.9
Q3.13 fixed	0.01	18.1	23.7	30.1	32.9	54.8	49.7	36.9
Variable	0.06	0.40	0.54	0.75	1.12	0.63	1.58	0.94

It is evident from the Table 4.2 that 16 bit floating point provides the best possible accuracy with respect to full precision inference. Q5.11 is the second best representation for this network. One must keep in mind that for other networks different QX.Y representations may be the ones with the best accuracy. However, due to the

[1]https://github.com/mvitez/thnets

approximately normal nature of parameter distribution, one of the representation will always be close in accuracy to full precision.

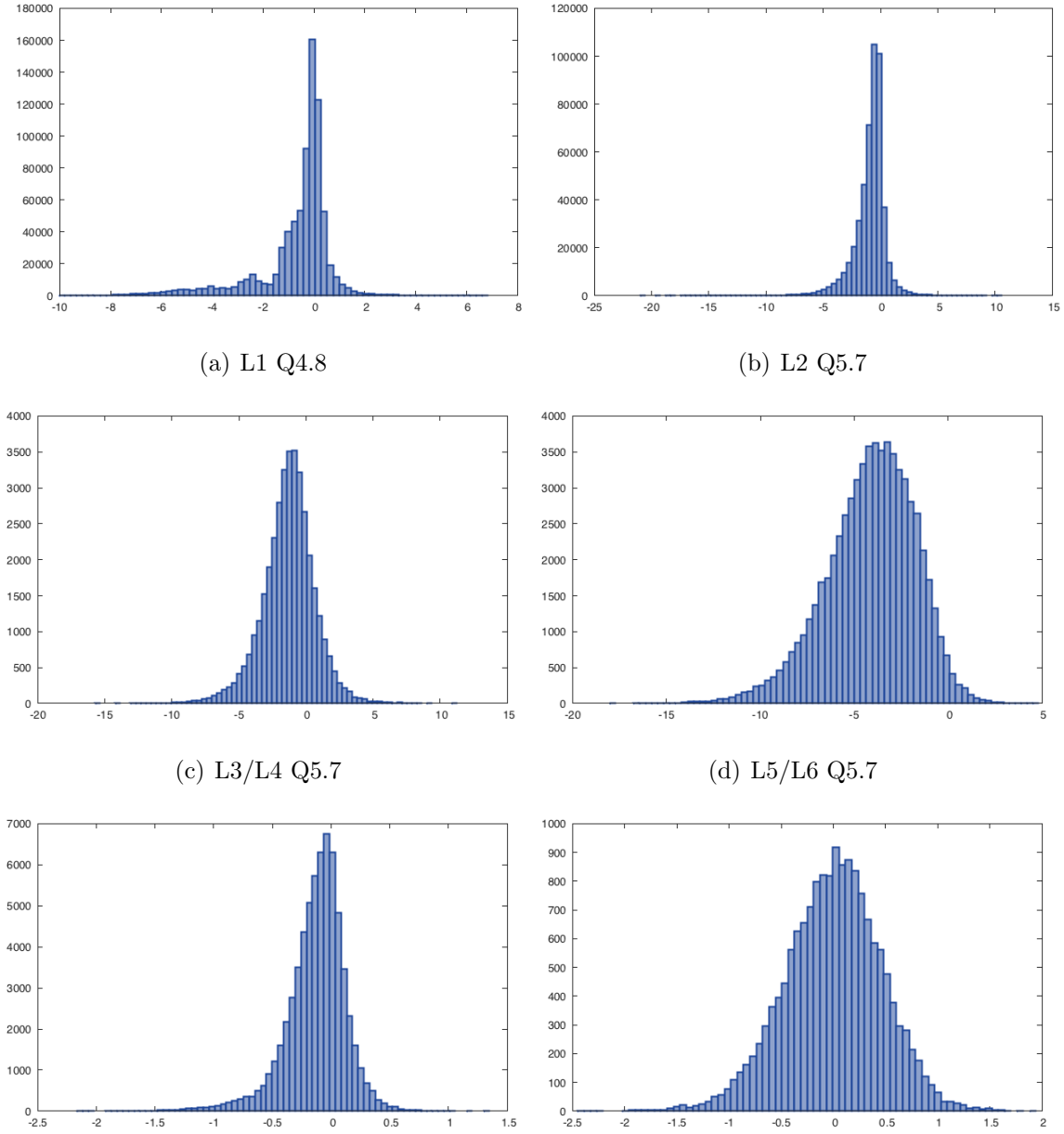
Variable Precision Fixed Point

The variable precision fixed point row in Table 4.2 corresponds to values obtained using Algorithm 1 from the previous chapter. This methodology was experimented for 8 bit and 12 bit numbers since with 16 bit numbers the Q5.11 representation is a clear winner. When using 8 bit fixed point for parameters and Q5.11 for feature maps, the results were fairly similar to the Q5.11 metrics. However, if both the parameters and the feature maps were shifted to 8 bit representation, error varied between 1% to 20% compared to full precision. The values mentioned in the Table refer to those obtained when using 12 bit representation for the parameters and feature maps. The variation in the feature map value range and the corresponding representation used is shown in the Figure 4.1.

4.3 Utilization

The utilization Figures 4.2 provided here are solely for the compute units. All figures except for 16bit floating point are for a 192 MAC system. In order to synthesize 192 16bit FP units, 57,408 LUTs are required which is greater than the available resources. The 8bit and 12bit systems utilize no DSPs. The 16bit system utilizes one DSP unit per MAC. The fixed point module utilizes 1 DSP per MAC and 12 additional DSPs for the output module increasing the total utilization to 204 DSP units. Additional resources are utilized for the DMA channels in and out of memory. When prototyped with Snowflake, the pipeline, caches, etc utilize additional resources too.

In order to obtain a better throughput we can clock the compute units much faster than other parts of the accelerator. However, we must be able to supply enough data to keep the units busy else the power increase will not be sufficiently offset by the



(a) L1 Q4.8

(b) L2 Q5.7

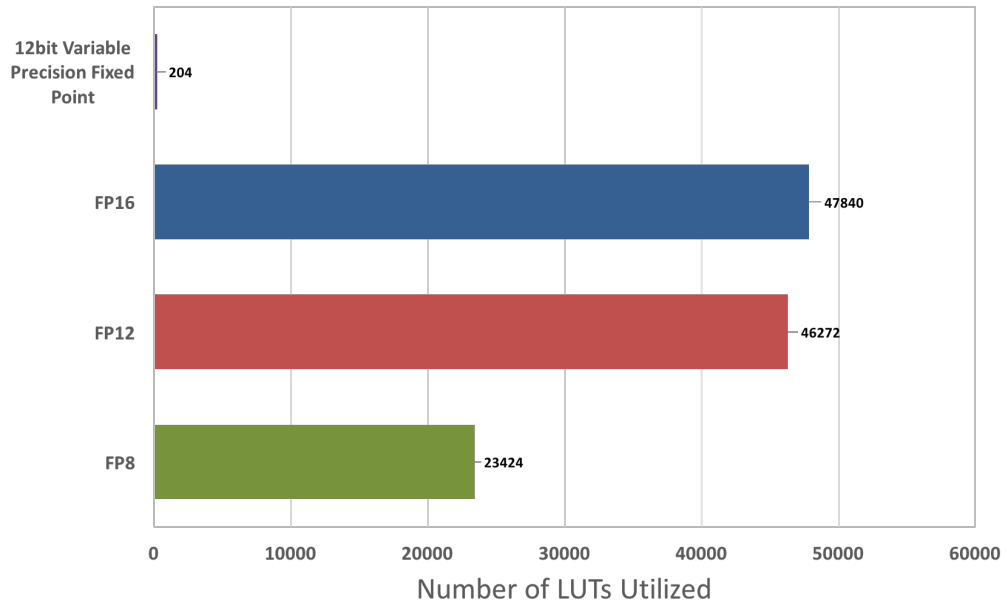
(c) L3/L4 Q5.7

(d) L5/L6 Q5.7

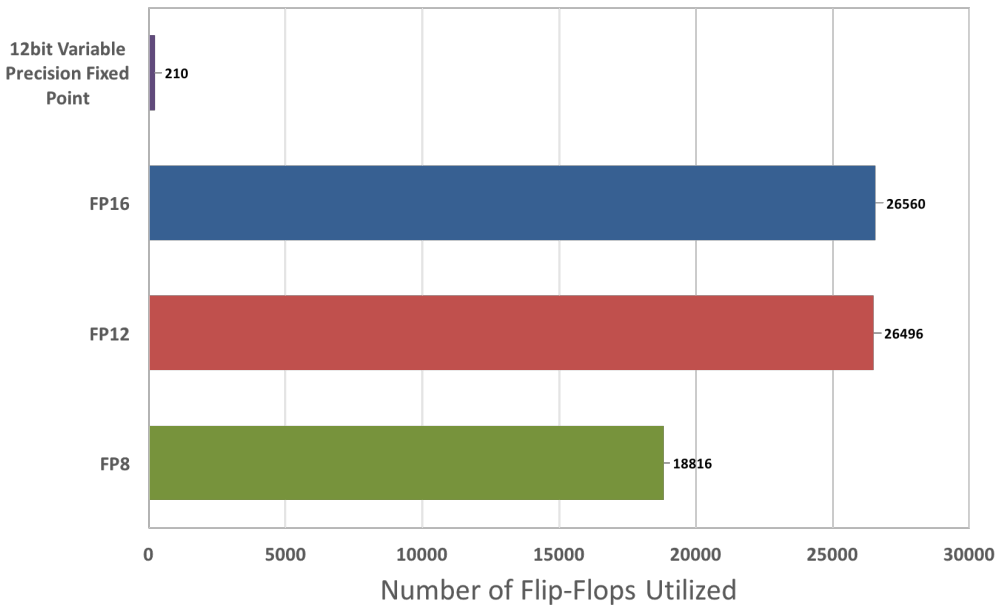
(e) L7 Q2.10

(f) L8 Q2.10

Fig. 4.1.: The distribution of the output feature map values are shown here. The first label presents the layer while the second label gives the representation to which these layers were truncated by the output unit



(a) LUT Utilization



(b) FF Utilization

Fig. 4.2.: Compute Unit utilization for various configurations. Note that all fixed point units are grouped into a single row. This is because even an 8bit dynamic fixed point MAC is sign extended to 16 bits in order to get it to synthesize as a DSP unit

throughput change. If we reduce the precision down to 12 bit variable fixed point, we find it much easier to sustain the bandwidth required for faster compute since the bandwidth required for 16 bit operands at 187.5 MHz is the same as that required for 12 bit operands at 250 MHz.

4.4 Performance

Figure 4.3 shows the performance per unit power for various number representations. One can observe that the Variable Precision Floating Point Representation provides a significant advantage over the 16bit FP representation while having comparable accuracy as seen in Table 4.2.

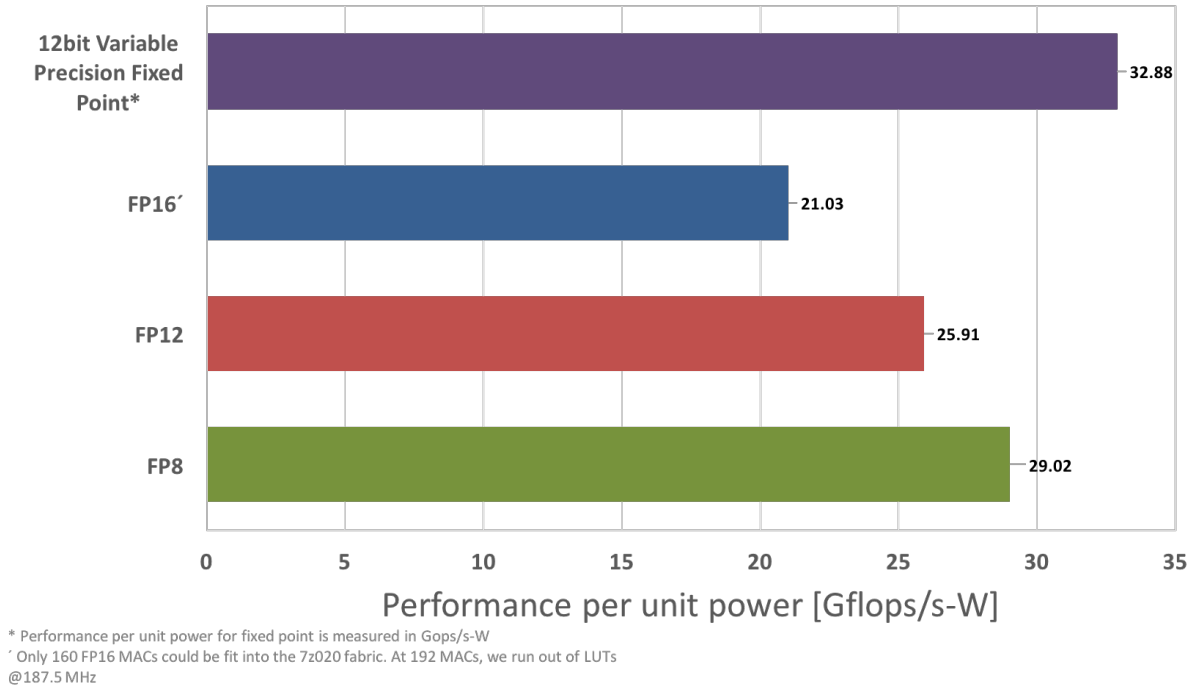


Fig. 4.3.: Performance per unit power for various representations (higher the better)

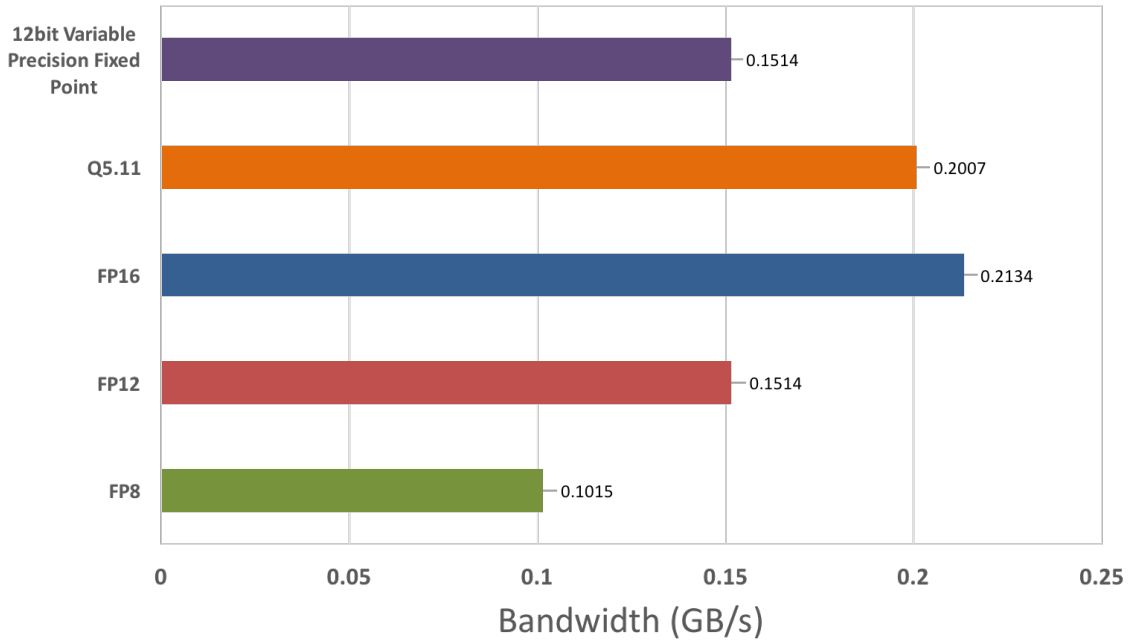


Fig. 4.4.: Bandwidth requirement for various representations (lower the better)

4.5 Bandwidth

Figure 4.4 shows the bandwidth requirement in Gigabytes per second for the various numeric schemes. It is evident that 12bit variable precision fixed point requires much lesser bandwidth than both Q5.11 fixed point and 16bit floating point representations. While 8bit FP presents the lowest bandwidth requirement, it degrades accuracy significantly.

5. ANALYSIS AND DISCUSSION

This chapter analyzes the results presented in the previous chapter and discuss the tradeoffs behind choosing the best possible implementation for the compute elements of a custom CNN architecture.

5.1 Accuracy/Utilization tradeoff

The 16 bit floating point unit has undoubtedly the best accuracy possible while the 12bit variable precision fixed point comes in a close second. However, in an embedded system accuracy is not the only parameter under consideration. A 16bit FP allows only 160 processing elements for every 192 elements for variable precision fixed point. If we were to choose the 16 bit FP, at optimum utilization it is capable of sustaining 54.22 Gflops/s while a fixed point unit will sustain 63.98 G-ops/s. Considering AlexNet with 2.4 G-ops per frame, 16bit FP provides a frame rate of 22.6 fps while variable precision fixed point provides 26.66 fps. While this may not be a big difference, one must also take into consideration the power savings due to lesser FPGA utilization and lower data transfer from memory. The fixed point units can also be boosted to double/triple the clock rate using multiple clock domains as described in Chapter 3. This will result in 1.5x-2.5x the original frame rate. The same can be done for the FP16 unit but it does not meet timing at higher frequency. At 375 MHz, the FP16 also utilizes greater fabric area and hence, synthesizing 160 units along with associated control logic is difficult.

5.2 Accuracy/Power tradeoff

Here, the 16bit floating point utilizes about 2.578 W power while the variable precision fixed point utilized 1.946 W. This jump in power gives an effective 21.03 G-ops/s-W for the FP16 and 32.88 G-ops/s-W for fixed point. This is quite a significant improvement for a 0.94% drop in accuracy.

5.3 Accuracy/Bandwidth tradeoff

Memory bandwidth is quite and important consideration in embedded systems. The memory bandwidth proves to be a bottleneck in quite a few embedded systems. Even if there is sufficient bandwidth available it is always better to access lesser data since it saves a significant amount of power. The variable precision fixed point representation requires significantly lower bandwidth compared to the 16bit fixed and floating point. Q5.11 fixed point representation and 16bit FP require more bandwidth due to the additional bits per transferred operand. We can see that 16bit FP requires more bandwidth than Q5.11. The reasons for this is the availability of lesser 16bit FP MAC units due to which feature Map data has to be transferred multiple times in order to compute the entire output. It is evident that 8bit FP has the lowest bandwidth but its poor accuracy is a downside that prevents us from considering this representation.

6. CONCLUSION AND FUTURE WORK

6.1 Future Work

This work identifies and tries to solve various accuracy performance tradeoffs in CNNs. It is limited because of its application solely for inference. Further development will revolve around implementing the same strategies in training.

Pruning unnecessary connection during training is showing promise. This pruning will reduce the range of numerical value of the parameters and feature maps even further allowing us to utilize lesser bits for representing each value while maintaining accuracy.

Current FPGAs are limited to a minimum of 16 bit computations. Research into next generation fabric with vector 8 bit multipliers will allow us to achieve even better performance. In this case the power consumption will have to be evaluated again in order to ensure that the benefit in performance is not offset by higher power draw resulting in a poorer G-ops/s-W metric.

6.2 Conclusion

In this thesis, we presented a case for experimenting with the compute elements in a CNN accelerator. The parameters and values in a trained CNN are usually normal distribution around a fixed value with a standard deviation. This allows us to use different number representations in order to extract the best accuracy from a layer at a low cost in term of power and bandwidth.

We then presented various strategies for improving the performance of a compute element while keeping the accuracy approximately constant. We described the various

units developed and presented an algorithm for extracting the best performance from each layer.

Next, we prototyped the proposed changes on a Zynq Zedboard and provided the various metrics obtained for the different strategies. Finally, we then tackled various tradeoffs for selecting the best possible compute unit implementation for a custom CNN architecture.

LIST OF REFERENCES

LIST OF REFERENCES

[1] A. Krizhevsky, I. Sutskever, and G. Hinton, "Imagenet classification with deep convolutional neural networks," *Advances in Neural Information Processing Systems*, vol. 25, 2012.

[2] R. Socher, B. Huval, B. Bath, C. D. Manning, and A. Ng, "Convolutional-recursive deep learning for 3d object classification," *Advances in Neural Information Processing Systems*, pp. 665–673, 2012.

[3] D. Ciresan, U. Meier, and J. Schmidhuber, "Multi-column deep neural networks for image classification," *Computer Vision and Pattern Recognition (CVPR), 2012 IEEE Conference on*, p. 36423649, 2012.

[4] J. Jin, A. Dundar, J. Bates, C. Farabet, and E. Culurciello, "Tracking with deep neural networks," *Information Sciences and Systems (CISS), 2013 47th Annual Conference on*, pp. 1–5, March 2013.

[5] Y. LeCun, L. Bottou, Y. Bengio, , and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, pp. 2278–2324, 1998.

[6] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," *CoRR*, vol. abs/1512.00567, 2015.

[7] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. E. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," *CoRR*, vol. abs/1409.4842, 2014.

[8] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *CoRR*, vol. abs/1512.03385, 2015.

[9] J. Long, E. Shelhamer, and T. Darrell, "Fully convolutional networks for semantic segmentation," *CoRR*, vol. abs/1411.4038, 2014.

[10] V. Badrinarayanan, A. Handa, and R. Cipolla, "Segnet: A deep convolutional encoder-decoder architecture for robust semantic pixel-wise labelling," *CoRR*, vol. abs/1505.07293, 2015.

[11] A. Paszke, A. Chaurasia, S. Kim, and E. Culurciello, "Enet: A deep neural network architecture for real-time semantic segmentation," *CoRR*, vol. abs/1606.02147, 2016.

[12] H. P. Graf, S. Cadambi, V. Jakkula, M. Sankaradass, E. Cosatto, S. Chakradhar, and I. Dourdanovic, "A massively parallel digital learning processor," in *Advances in Neural Information Processing Systems 21* (D. Koller, D. Schuurmans, Y. Bengio, and L. Bottou, eds.), pp. 529–536, Curran Associates, Inc., 2009.

[13] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, (New York, NY, USA), pp. 269–284, ACM, 2014.

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

[15] J. Cloutier, E. Cosatto, S. Pigeon, F. R. Boyer, and P. Y. Simard, "Vip: an fpga-based processor for image processing and neural networks," in *Microelectronics for Neural Networks, 1996., Proceedings of Fifth International Conference on*, pp. 330–336, Feb 1996.

[16] C. Farabet, B. Martini, B. Corda, P. Akselrod, E. Culurciello, and Y. LeCun, "Neuflow: A runtime reconfigurable dataflow processor for vision," *Proc. Fifth IEEE Workshop Embedded Computer Vision*, 2011.

[17] V. Gokhale, J. Jin, A. Dundar, B. Martini, and E. Culurciello, "A 240 g-ops/s mobile coprocessor for deep neural networks," in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*, June 2014.

[18] A. Putnam, A. Caulfield, E. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, E. Peterson, A. Smith, J. Thong, P. Y. Xiao, D. Burger, J. Larus, G. P. Gopal, and S. Pope, "A reconfigurable fabric for accelerating large-scale datacenter services," June 2014.

[19] Q. Z. Wu, Y. L. Cun, L. D. Jackel, and B. S. Jeng, "On-line recognition of limited-vocabulary chinese character using multiple convolutional neural networks," in *Circuits and Systems, 1993., ISCAS '93, 1993 IEEE International Symposium on*, pp. 2435–2438 vol.4, May 1993.

[20] S. RAJASEKARAN and G. PAI, *NEURAL NETWORKS, FUZZY LOGIC AND GENETIC ALGORITHM: SYNTHESIS AND APPLICATIONS (WITH CD)*. PHI Learning, 2003.

[21] T. M. Bartol, C. Bromer, J. P. Kinney, M. A. Chirillo, J. N. Bourne, K. M. Harris, and T. J. Sejnowski, "Hippocampal spine head sizes are highly precise," *bioRxiv*, 2015.

[22] Z. Deng, C. Xu, Q. Cai, and P. Faraboschi, "Reduced-precision memory value approximation for deep learning," *HPL-2015-100*, 2015.

[23] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, "Deep learning with limited numerical precision," *CoRR*, vol. abs/1502.02551, 2015.

[24] S. Han, J. Pool, J. Tran, and W. J. Dally, "Learning both weights and connections for efficient neural networks," *CoRR*, vol. abs/1506.02626, 2015.

[25] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding," *CoRR*, vol. abs/1510.00149, 2015.

[26] Xilinx, *AXI Interconnect v2.1.* Vivado Design Suite, November 2015.

[27] Avnet, *Zedboard Product Brief.* November 2014.

[28] R. Collobert, K. Kavukcuoglu, and C. Farabet, "Torch7: A matlab-like environment for machine learning."