Reconfigurable Low Arithmetic Precision Convolution Neural Network Accelerator VLSI Design and Implementation

En-Ho Shen

Advisor: Shao-Yi Chien

Graduate Institute of Electronics Engineering

National Taiwan University

Taipei, Taiwan

July 2019

Abstract

Deep neural networks (DNNs) shows promising results on various AI application tasks. However such networks typically are run on general purpose GPUs with bulky size and hundreds of watt power, unsuitable for mobile applications. In this thesis, we present a VLSI architecture able to process on quantized low numeric precision convolution neural networks (CNNs), cutting down on power consumption from memory access and speeding the model up with limited area budget, particularly fit for mobile devices. We first propose a quantization re-trainig algorithm for trainig low-precision CNN, then a dataflow with high data reuse rate with a specially data multiplication accumulation strategy specially designed for such quantized model. Such data requires specially designed arithmetic unit for its full potential, we design a micro-architecture for low bit-length multiplication and accumulation, then a on-chip memory hierarchy and data re-alignment flow for power saving and avoiding buffer bank-conflicts, and a PE array designed for taking broadcast-ed data from buffer and sending out finished data sequentially back to buffer for such dataflow. The architecture is highly flexible for various CNN shaped and re-configurable for low bit-length quantized models. The design synthesised with a 180KB on-chip memory capacity and a 1340k logic gate counts area.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Deep neural networks (DNNs) have shown promising capability in numbers of AI applications, including computer vision, natural language processing and even gaming. The performance of DNN is improving rapidly: the ImageNet classification challenge has surpassed human-level top-5 accuracy 95.51% (ResNet-152)[9] and even beyond.

However, such performance requires tens to hundreds of parameters, up to billions of operations for a single image inference. For example, AlexNet [10] takes 1.4GOPS to process a single 224x224 image, while ResNet-152 takes 22.6GOPS, more than an order of magnitude more computation. In order to run these model in real time, modern powerful general-purpose GPUs are mandatory, indicating deployment of said models on embedded devices, where the true potential of artificial intelligence lies, impractical.

Besides processing speed, the enormous amount of operations and memory transactions introduce unbearable energy consumption for mobile devices. The energy cost per 32b operation in a 45nm technology ranges from 3pJ for multiplication to 640pJ for off-chip memory access [11]. To run a billion connection neural network at 30 FPS would require 30Hz*1G*640pJ = 19.2W just for DRAM accesses, well beyond the several hundreds mini Watt to couple of Watt range of typical mobile battery-powered devices.

To address the problem, we propose a re-configurable accelerator hardware, to efficiently run linearly-quantized CNN model, saving both computational and memory transaction power from the micro-architectural, system-level dataflow, algorithmic quantization strategy perspectives.

1.1 Motivation

Researchers have dedicated to either optimized model or dedicated hardware for DNN. For algorithmic optimizations,some would compress pre-trained deep networks taking advantage of the sparse property of DNN, even further encode the final model [2],[12]; Some trim their model through pruning, discarding unwanted or insignificant weights. [2],[13],; there are researches that directly re-design the networks, seen in [14],[15], often replace the original computational dense regular convolution layer with group convolution or depth-wise convolution, also they insert 1x1 filter between normal 3x3 filters, drastically reduces the input channel size to following convolution layer with an additional layer of non-linearity. These approaches introduce certain degrees of irregularity to the computation. For compression, encoded weights will ultimately be decompressed to be used in computation, as seen in [11], [12], putting extra burden on the processors, but often cut down on the data transaction cost. For specially designed mobile friendly models, they are often memory-bound, requiring large bandwidth on off-chip memory, which would not be easily optimized without system-level design, out of the main context of hardware accelerator design of this thesis.

With the main focus of low-power and above concerns in mind, we find that linear quantisation of deep models on both weights and activations is a rather regular compression strategy, reducing memory traffic without complex indexing and decompression on the accelerator side. However, common processions are usually only equipped with 8/16 bit and floating point adder and multiplier, deploying low-precision model onto normal processor will be a waste in micro-architectural

perspective. Besides, ultra low-precision (under 8 bit) has been shown to pose great drop on accuracy applied to the each layers of a model on large-scale dataset like ImageNet. Workarounds have been seen in many works, including leave the most information-rich layers, which are the first convolution layer and the last fully-connected layer, untouched [3]. It is also shown [3],[16] that different layers of a deep model response differently to quantisation: more bit for all resulting waste of resources, less bit for all ended up model accuracy loss.

Therefore, a flexible accelerator for low-bit arithmetic is mandatory. First, we propose a simple trick on training low-precision deep network, exploiting the potential of flexible bit-width quantization on AlexNet, ImageNet dataset. Seeing the approach working, we then design a dataflow architecturally for low-bit arithmetic operations, and we also have to design a re-configurable arthmetic-logic unit capable of computing 1,2,4,8 bit addition and multiplication.

1.2 Contribution

In this thesis, we seek to fully explore the benefit adopting low-precision arithmetic operations, design and implement an accelerators capable of processing compacted low-precision data which is otherwise incompatible to modern GPUs and CPUs. We start off with an algorithm determining the appropriate clipping threshold for quantizing deep neural network, achieving promising **54%** accuracy on large image recognition task dataset **ImageNet**[17] on **AlexNet**[10] using both quantized weights and activations with no more than 4 bits except for the 8-bits input and last full-precision fully-connected layer. This experiment supports the feasibility of a dedicated hardware operating on such network. We proceed to design and implement a flexible re-configurable accelerator, designed for ***subword accumulation*** arthemtic operation, with 5 modes being: 1) 16 1-bit XNOR operation; 2)16 1-bit MAC (multiply and add) operation; 3) 8 2-bits MAC; 4) 4 4-bits MAC; 5) 2 8-bits MAC. Except for the XNOR operation, unsigned and signed operation

for 1-8 bits data are supported in order for computing data of larger bit counts using lower bits operation. The core of our processing array is composed of 16x16 PEs (processing element), one 8KB weight buffer, one 16KB input buffer and a 100KB global buffer; inside each PE column there are 1 12x16b, 16 48x16b and 16 32x32b register files. The four-level memory hierarchy keeps data in lower and smaller memory hierarchy longer for the best power consumption saving. We propose a ***output row stationary*** dataflow that each PE takes charge of one output partial sum row at a time, exploiting convolutional data reuse to the maximum. The system synthesized to 1340K logic gates area and 180KB on-chip memory, capable of processing AlexNet on ImageNet quantized to 4-bit with a fast 206.9FPS and dissipates 573mW averagly. We show promising results on quantization CNN with both accuracy and performance, the entire process is entitled to be a standard for CNN deployment onto mobile devices.

1.3 Thesis Outline

In this chapter 1, we briefly go through some of the works striving for neural network simplification, and bring out the mindset that drives us to work on model quantisation and re-configurable accelerator design. The related works are mentioned in chapter 2. Knowledge regarding convolution neural network and quantisation is in chapter 3, a quantisation method is also proposed. Moving on to the hardware design, we'll discuss about the proposed architecture and specification in chapter 4. The implementation and performance results are in chapter 5. At last, conclusion is given in chapter 6.

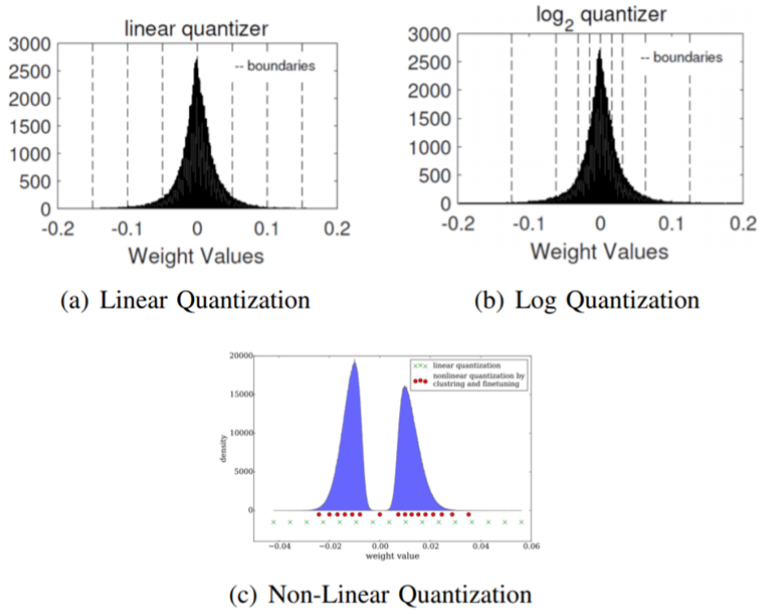
Chapter 2

Related Work

Many approaches have been proposed aiming for deploying DNN on mobile device, either in algorithm or hardward aspect. In this chapter, we will discuss more about the main optimization scheme we choose, which is quantisation, and then existing DNN accelerators hardware design.

2.1 Quantization

Quantization maps data to a smaller set of quantization levels. The principle is to minimize the error between the reconstructed data from the quantized one and the original data. The quantization level essentially reflects the bits required to represent the data. Reduced data bits comes with several benefits including reduced storage costs, memory transactions and computational cost. There are ways to quantization, from simplest uniform distance between each quantization Figure 2.1(a) level to special mapping function such as *log* function so that distance between quantization steps are of logarithm relation Figure 2.1(b); and even more advanced approaches like clustering data into groups with k-means, requiring look up table for the mapping and computation Figure 2.1(c). The simplest quantization also known as *linear quantization* related closely to this project, and we're going to focus on it particularly.



(a) Linear Quantization (b) Log Quantization

(c) Non-Linear Quantization

Figure 2.1: Various methods of quantization; taken from [1][2]

2.1.1 fixed point quantisation

More related to this work, researches find the numerical requirement for even the latest DNN model inference stage far from the commonly used 32/64 bit floating-point format. [16] quantize the models to fixed point with L_2 error minimization, achieved **0.89%** MCR (miss classification rate) on **MNIST**[18] with 5 bits weights in comparison to **0.81%** MCR with floating point weights.

2.1.2 ternary to binary quantisation

Some researchers have tried extreme quantization down to 2bits ternary weights such as in TWNs[19] and even binary weights in BinaryNet [20], to binarize both weight and activation as in XNOR-net[3].

TWNs minimizes the Euclidian distance between the full precision weights $\boldsymbol{W}$ and ternary-valued weights $\boldsymbol{W}^t$ along with a non-negative scaling factor α in (2.1) achieving **99.35%**, **92.56%**, **84.2%** on **MNIST**, **CIFAR-10**[21], **ImageNet(top-**

5) dataset respectively.

$$\begin{aligned} \alpha^*, \boldsymbol{W^{t*}} &= \arg\min_{\alpha, \boldsymbol{W^t}} = \|\boldsymbol{W} - \alpha \boldsymbol{W^t}\|_2^2 \\ \text{s.t } \alpha \geq 0, \boldsymbol{W_i^t} &\in \{1, 0, -1\}, i = 1, 2, ..., n. \end{aligned} \tag{2.1}$$

BinaryNet constrains both weights and activations to either +1 or -1 with

$$x^b = \text{Sign}(x) = \begin{cases} +1 & \text{if } x \geq 0, \\ -1 & \text{otherwise}, \end{cases} \tag{2.2}$$

achieving **0.96%** , **10.15%** error rate on **MNIST** , **CIFAR-10** dataset respectively. And finally the Xnor-net put their emphasis on the scaling factor α^* and $\boldsymbol{K}$ between layers that first computes the floating point value of an output tensor, then take the sign of it for the binarized activation for the input of next layer.

$$\begin{aligned} \alpha^* = \frac{\boldsymbol{W^T} \text{sign}\,(\boldsymbol{W})}{n} = \frac{\Sigma \|\boldsymbol{W_i}\|}{n} = \frac{1}{n}\|\boldsymbol{W}\|_{l1} \\ \boldsymbol{K} = \frac{\Sigma \|\boldsymbol{I}_{:,:,i}\|}{c} * \frac{1}{w \times h} \end{aligned} \tag{2.3}$$

In a Xnor-net particularly the convolutional operation seen in Figure 2.2 can be appriximated by:

$$\boldsymbol{I} * \boldsymbol{W} \approx (sign(\boldsymbol{I}) * sign(\boldsymbol{W})) \odot \boldsymbol{K}\alpha \tag{2.4}$$

where $\boldsymbol{I}$, $\boldsymbol{W}$ denotes the input, weight tensors of a layer. Additionally, they discovered that regular batch-normalization layer position between convolution and activation function is unfriendly to binarization algorithm, by moving convolution layer after the activation function instead further boost the accuracy, achieving **69.2%** top-5 accuracy on **ImageNet** using **AlexNet** model. By adopting binarization scheme, XNOR and bitcount operations can be applied to save computational cost at inference stage, with **58x** reported speed up against CPU time.

Pushing the limit to binarization hurts the accuracy by a large margin, worth mentioning that in Xnor-net they didn't quantize the first and the last layer of the convolution, which bear too much information to be discarded.

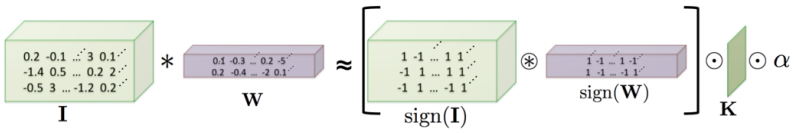


Figure 2.2: Convolution with XNOR-Bitcount in XNOR-net; taken from [3]

2.1.3 8-bit quantization on modern models

More practically, modern deep models have been showed to work on 8 bits quantization on both weights and activations [4]. Taking advantage of Nvidia's bulit-in INT8 operations on their GPUs, they developed a CUDA library in TensorRT with an algorithm minimizing loss of information when quantizing trained model without further fine tuning or retraining. Starting with the tensor approximation with quantized tensor:

$$\textbf{Tensor Values} \approx \textbf{FP32 scaling factor} * \textbf{INT8 array} \tag{2.5}$$

The scaling factor determines the mapping of the quantized level to the original value, for example a scaling factor of $\frac{127}{2.7}$ maps the original data 2.7 to quantized data 127. This is always a trade-off process between range and precision: the larger the scaling is, the larger the range, and the smaller the precision. The scaling process also includes the clipping of data exceeding a pre-defined threshold as seen in Figure 2.3, and the threshold is in fact the scaling factor times the maximum level of the quantization, which is 127 in INT8 case.

Therefore the main challenge here is the choice of the scaling factor.As stated in the work, INT8 model encodes the same information as the original FP32 model, the choosing of the scaling factor is a process of minimizing the loss of information, which can be measured by Kullback-Leibler divergence as the relative entropy or information divergence (2.6) between two discrete probability distributions $\boldsymbol{P}$, $\boldsymbol{Q}$. They propose a **Calibration Dataset** to be run on FP32 format inference, collecting histograms of activations, and then pick threshold that minimize the KL divergence by generating many quantized data distributions with different

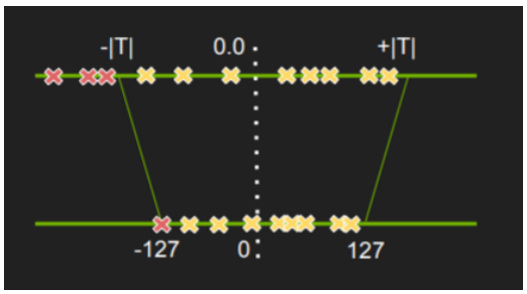


Figure 2.3: Information loss is minimized through careful choice of saturation threshold; taken from [4]

saturation thresholds. This process is said to take only a few minutes on a desktop workstation.

$$\textbf{KL\_Divergence}(\boldsymbol{P}, \boldsymbol{Q}) := \textbf{SUM}(\boldsymbol{P}[i] * log(\frac{\boldsymbol{P}[i]}{\boldsymbol{Q}[i]}), i) \tag{2.6}$$

The resulting INT8 model achieves no more than **0.46%** extra error rate if even not lesser, at the same time speeding up from **1.62x** to **3.67x** depending on the inference batch size (from 1 to 128) on their processor DRIVE PX2, dGPU.
In chapter 3 we propose a similar yet simpler scheme to search for the desirable quantization thresholds; although our approach involves no calibration after the training, it requires re-training on low-precision inference, and is likely to be slower than train-then-fine-tune scheme in this work. However, we believe that re-training on ultra-low-precision model (< 8 bits) is still necessary.

2.2 Hardware design

Recently researches related to deep neural network accelerator are blooming rapidly. They typically start off with a core algorithm simplifying the NN model, then combining either micro-architectural optimization such as low-precision arithmetic units or system level optimization involving data compression between the chip and the off-chip memory, delicate buffer design, sparsity-aware zero-skipping operations and so on.

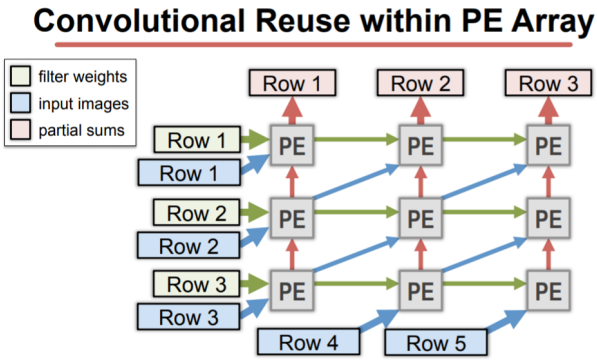


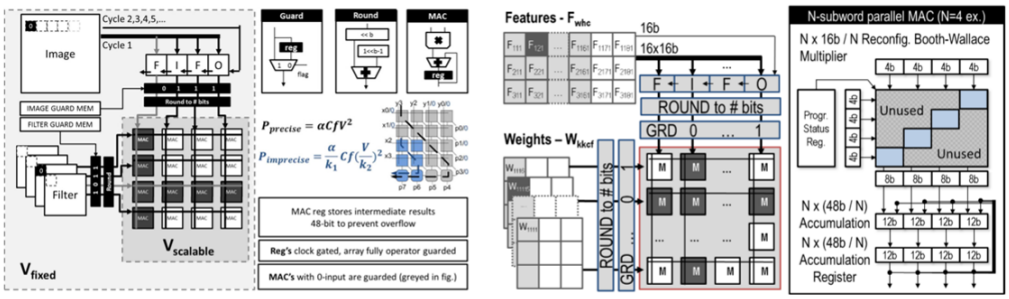
Figure 2.4: Row stationary: rows passed to processing elements as unit; taken from [5]

We are going to focus on several works that inspire us the most, including modified classic systolic-array style processor operating on a granularity of rows, SIMD (single instruction multiple data) style processor coupled with re-configurable arithmetic unit being able to operate on 1-16 bits respectively. From these works, we conclude that slashing down memory access counts either from on-chip or off-chip is of upmost importance for chip power efficiency, through further low-bit operation optimization the memory transaction frequency can be furthered improved.

2.2.1 Dataflow optimization: row stationary

Eyeriss[12] analyzes existing accelerator work in a now widely used manner, classifying them into *Weight Stationary*, *Output Stationary*, *No Local Reuse* three dataflows that reuse data differently, and propose a novel processing dataflow coined *Row Stationary* as in Figure 2.4.

The dataflow finely captures all sources of data reuse opportunities ranging from convolution reuse, batching, filter reuse. The work also adopts various optimization methods such as zero-gated register file, running-length coded data in-and-out of the chip, dynamic voltage and frequency scaling and so on. The work consisting of 168 PE with 16-bit fixed-point multipliers tapes out to have a peak performance of



(a) (b)

Figure 2.5: (a) Precision re-configurable multiplier and (b)Sub-word parallelism re-configurable multiplier; taken from [6][7]

42 GMACS, operating avergely on **278mW**.

2.2.2 Precision reconfigurable and sub-word parallelism arithmetic unit

[6] is one of the pioneering re-configurable low-precision processor for DNN. The 16-bit multiplier in each PE can be re-configured into one 1-16 bit multiplier, the resulting switching activity and critical path then scale with the configuration, allowing possible lower power supply at constant frequency according to power estimation equation $\boldsymbol{\alpha C f V^2}$ as in Figure 2.5 (a), the unused logics are gated accordingly to prevent unwanted power dissipation. In terms of dataflow, the weights are shared along a PE row, the inputs are shared along a PE column in contrast; worth noting that a large shifting FIFO for input data is implemented so it propagates input data to adjacent PE column, this also indicates that the PE adopts *output stationary* scheme. The work has a peak performance of **102 GOPS**, operating on **76mW** (AlexNet) and a maximum of **2.6TOPS/W** power efficiency.

[7] inherits from [6]. Instead of a multiplier operating on 1 pair of variable bit-length data, this work proposes multiplier with $\boldsymbol{N}$=1,2,4 sub-word parallelism, in other words, a maximum of $\boldsymbol{N}$x peak performance as in Figure 2.5 (b). The

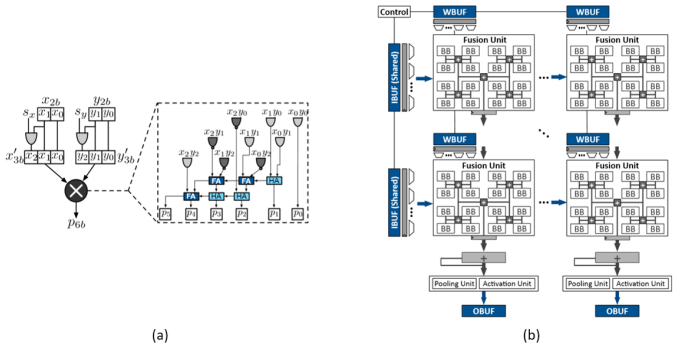


Figure 2.6: (a)BitBricks computing ternary multiply and add (b)Systolic-array processing architecture; taken from [8]

work has a peak performance **102N*GOPs**, operating from **7.5-300mW** and a maximum of **10TOPS/W** power efficiency.

By *output stationary* scheme indicates both works above captures less convolutional reuse, and having sub-word parallelism means large local output register which renders row stationary dataflow inapplicable.

2.2.3 Bit-level re-configurable arithmetic unit

[8] bears the most similarity to our work. A atomic processing block *BitBricks* is proposed, which is effectively a ternary (-1,0,+1) mutliply and add unit with a 6b output as in Figure 2.6 (a). 16 *BitBricks* are put together inside one *Fusion Unit*, offering 1,2,4,8,and 16 fused-BitBricks with varying operand bitwidths, output bitwidth and level of parallelism. The *Fusion Units* are arranged in systolic-array manner, propagating partial sum vertically to output register and buffer as in Figure 2.6 (b). The synthesized result are used to conducted extensive experiments comparing against other works. They claim to offer **3.9x** speed up and **5.1x** energy saving over [12] under same area, frequency and process technology; also they claim to match the performance of 250W Titan Xp using INT8 vector instructions, while consuming only **896mW**.

To have the best flexibility as this work does, an 6b output 2b multiplier would be inefficient area-wise to some degree, also considering a 16 *BitBrick* fused 8-bit multiplier is essentially a 16 6-bit multiplier-adder tree, possibly introducing large area overhead in comparison to a modern 16-bit booth algorithm multiplier.

Works from above provide us several great starting points, involving *row stationary* dataflow, re-configurable low-precision multiplier-add unit, SIMD architecture design.

Chapter 3

Low numeric precision convolution neural network

In this chapter, we go through the basic of convolution neural network, the mindset behind choosing linear quantization scheme, an algorithm for training a ultra-low precision neural network on large-scale dataset and finally a tensor re-pack method for deploying the network on a processor supporting low-precision MAC operation.

3.1 Convolutional Neural Networks

A typical convolution neural network layer takes a 3D *input tensor* and a *filter tensor*, and performs 2D convolution, producing a 3D *output tensor*.It is essentially summing up **C** channel of 2D convolution for one output pixel as shown in Figure 3.1, this process is repeated for **M** number of filter, resulting in output tensor of **M** number of channel. In Figure 3.2 shows a common techniques *batching*, taking multiple input tensor and perform convolution on them at the same time; the picture also provides an example of the shape parameter **U**. The entire process

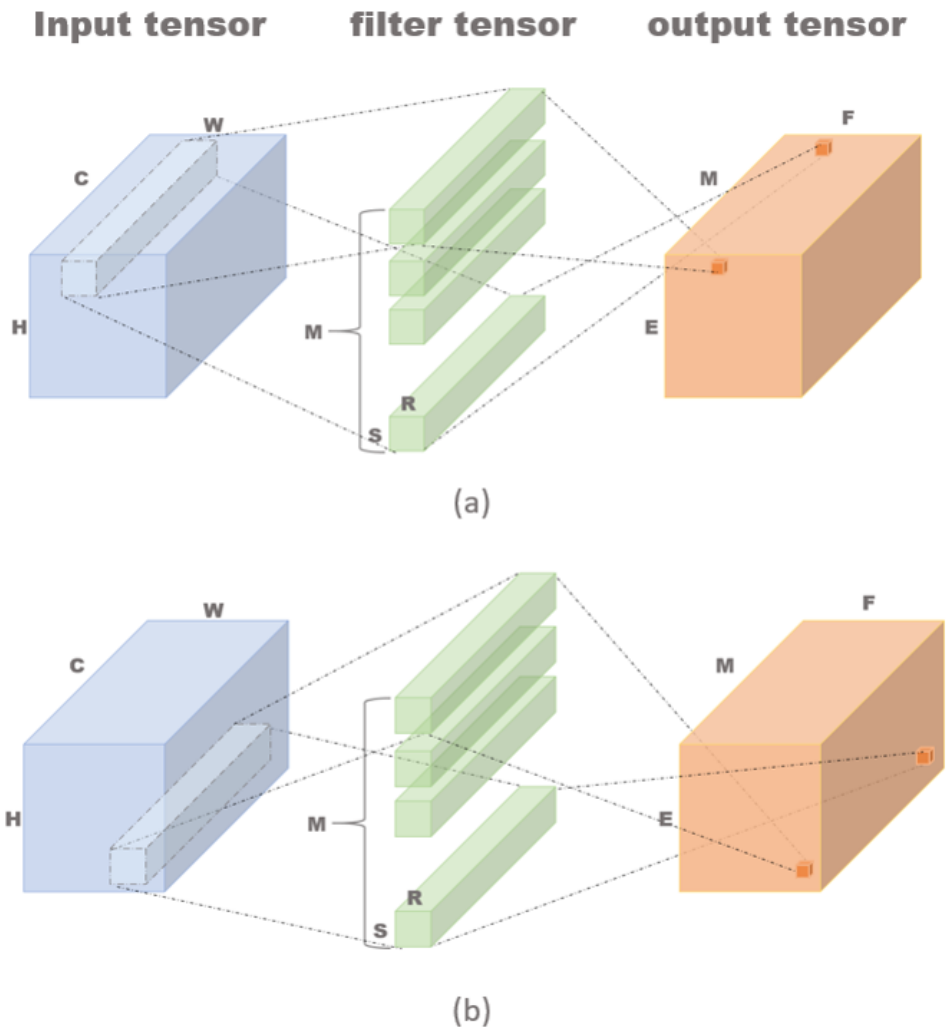


Figure 3.1: Computation of a convolutional layer.

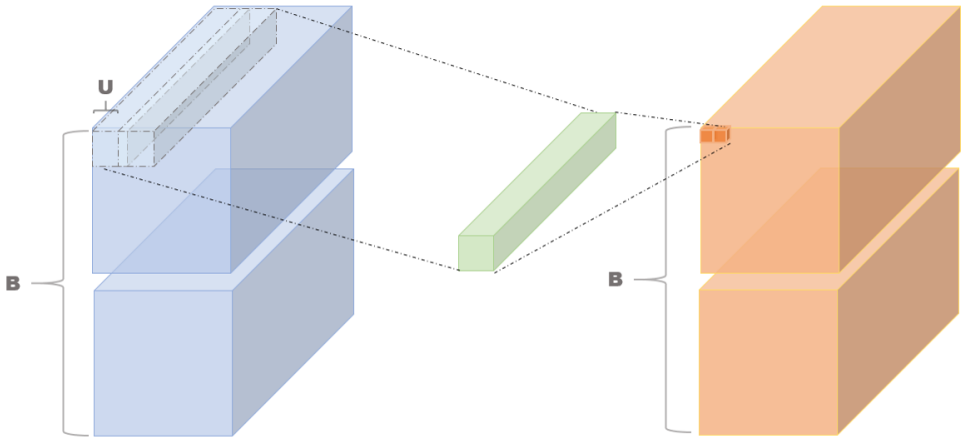


Figure 3.2: Shape parameter B and U.

Table 3.1: Basic shape parameters of a CNN layer

Parameter	Description
H/W	Input feature map spatial dimensions
E/F	Output feature map spatial dimensions
R/S	Filter spatial dimensions
C	Input channels
M	Output filters
U	Convolution stride
B	Batch,# of feature maps to be processed

is formulated by Equation 3.1:

$$\boldsymbol{O}[z][u][x][y] = \text{ReLU}\,(\sum_{k=0}^{C-1}\sum_{i=0}^{R-1}\sum_{j=0}^{S-1}\boldsymbol{I}[z][k][Ux+i][Uy+j] * \boldsymbol{W}[u][k][i][j]),$$
$$0 \leq z \leq B\,, 0 \leq u \leq M\,, 0 \leq y \leq E\,, 0 \leq x \leq F,$$
$$E = (H - R + U)/U,\ F = (W - S + U)/U \tag{3.1}$$

3.2 Low Precision CNN

By choosing linear quantization regular multiplier and adder can be used unlike the need of look-up table in non-linear quantization. As mention briefly before, the main benefit of quantization is reduction in memory transaction, especially off-chip DRAM access. We will show in chapter 5 that given the same dataflow, 8-bit quantized model off-chip memory access wins over 16-bit model compressed with *running length coding* with data sparsity of more than 70%, indicating that quantization is a reliable compression scheme independent of sparsity of the data. This sums up our reasons behind choosing linear quantization.

Low numeric precision data in this work is applied to each type of tensors. Given a quantization bit $\boldsymbol{k}$, clipping threshold $\boldsymbol{\tau}$, tensor $\boldsymbol{D}$, the quantized tensor $\boldsymbol{D_q}$ is given by Equation 3.2:

$$\begin{aligned} \boldsymbol{D_q} &= round\ (\ \frac{2^{k-1}}{\boldsymbol{\tau}} * clip\ (\boldsymbol{D}, \boldsymbol{\tau})\ ) \\ clip(\boldsymbol{D}, \boldsymbol{\tau}) &= \begin{cases} \boldsymbol{D} & \|\boldsymbol{D}\| < \boldsymbol{\tau} \\ sign(\boldsymbol{D}) * \boldsymbol{\tau} & \text{otherwise} \end{cases} \end{aligned} \tag{3.2}$$

Where *round*() round the number to the nearest integer. The convolution on the original tensor can therefore take the following form Equation 3.3:

$$\begin{aligned} \boldsymbol{O} \approx \boldsymbol{I} \star \boldsymbol{W} = \boldsymbol{I_q} \star \boldsymbol{W_q}\ \frac{\boldsymbol{\tau_I}}{2^{k_I-1}}\ \frac{\boldsymbol{\tau_W}}{2^{k_W-1}} &= \boldsymbol{O_q}\ \frac{\boldsymbol{\tau_O}}{2^{k_O-1}} \\ \boldsymbol{\alpha} \boldsymbol{I_q} \star \boldsymbol{W_q} &= \boldsymbol{O_q} \end{aligned} \tag{3.3}$$

$$\boldsymbol{\delta} = round(log_2 \alpha) \quad \boldsymbol{\gamma} = round(\frac{\alpha}{2^{\delta}}) \quad \boldsymbol{\alpha} \approx= \delta \gg \gamma \tag{3.4}$$

We see the part $\boldsymbol{I_q} \star \boldsymbol{W_q}$ is the low precision convolution fit for specially designed processor. $\boldsymbol{\alpha}$ merges the rest of the equation to a scaling factor, which in practice can be approximated together with other scaling factor with a 16-bit fixed-point multiplier $\boldsymbol{\delta}$ and a shifting factor $\boldsymbol{\gamma}$, see Equation 3.4.

The training of low precision neural network follows that of [3]. See algorithm 1; at training, the quantized convolution results are fed forward through the network,

error is computed based on the quantized value, and the gradient is back-propagated bypassing the quantization steps, updating the full-precision weights.

Algorithm 1: quantization NN training

1 $\boldsymbol{O} = \textbf{QuantForward}(\boldsymbol{I_q}, \boldsymbol{W_q}, \boldsymbol{\tau_I}, \boldsymbol{\tau_W}, k_W, k_I)$

2 $\frac{\partial C}{\partial \boldsymbol{W}}$ = **QuantBackward**($\frac{\partial C}{\partial \boldsymbol{O}}$, $\boldsymbol{W_q}$) // the gradient is caculated based on the quantized weight

3 $\boldsymbol{W^{t+1}} = \textbf{UpdateParameters}(\boldsymbol{W^t}, \frac{\partial C}{\partial \boldsymbol{W}})$ // real gradient is replaced by that from quantized weight, bypassing the quantization function

3.3 Quantization Loss Minimization Threshold Selection

We propose a simple algorithm for quantization threshold choosing. *Batch Normalization* layer [22] has been widely used in modern NN models, speeding up the convergence process and possibly improve the accuracy. The layer introduce additional scaling factor and bias base on input data distribution along the batch, it helps the *internal covariate shift* phenomena, so that non-linearity layers take effect more reliably. Simply put, the scaling and bias calibrate the output to have a mean of 0 and variance of 1, and naturally, it is heuristic to model the distribution with a *standard normal distribution*. With the assumption that our data are of standard normal distribution, given a target quantization bit, we would generate a large number of data of standard normal distribution, iterate over a set of threshold selection, and pick the threshold posing the least quantization error, the error is either calculated with L_1 norm or L_2 norm. The thresholds for each quantization bits are then fixed and applied throughout the entire model. We tested the impact of error calculation on model accuracy and choose L_1 norm.

Algorithm 2: quantization threshold selection

Input: $k, \boldsymbol{\tau}$

Output: τ

generate $\boldsymbol{D}$ array of standard normal distribution array

1: **for** each item τ_i in $\boldsymbol{\tau}$ **do**

2: $\quad \boldsymbol{D_q} = round\ (\ \frac{2^{k-1}}{\tau_i}\ clip\ (\boldsymbol{D}, \tau_i)\ )$

3: $\quad \boldsymbol{D_r} = \boldsymbol{D_q} * \frac{\tau_i}{2^{k-1}}$

4: $\quad \boldsymbol{E_{1i}} = \Sigma \| \boldsymbol{D} - \boldsymbol{D_r} \|$

5: **end for**

6: **return** $\tau_i = \underset{\tau_i}{\operatorname{argmin}} \boldsymbol{E_1}(\tau_i)$

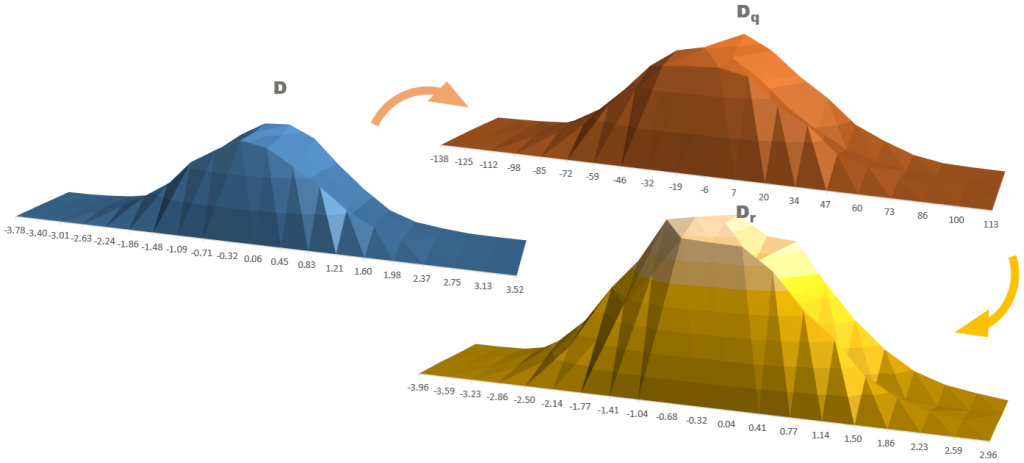


Figure 3.3: Given a threshold, calculate the error between reconstructed distribution and the original.

Table 3.2: Threshold for each bit setting by L_1 norm error

Bit length	1	2	3	4	5	6	7	8	32
Threshold	1.2	1.9	2.4	2.7	3.1	3.5	3.6	3.7	100

3.4 Computational consideration and data re-packing

Now that we have the quantized model, we need efficient computational strategy. Simply store the model, say quantized to 4 bits, in original 32-bit data type is 8x waste of storage, memory bandwidth and can't be effectively deployed to processing unit. We propose a simple data re-packing method, to tightly compact the data. However the compacted data requires specially designed processor to reach its full performance potential. Since we're aiming at large image recognition task, the data usually can't fit onto the accelerator entirely. We use several tiling parameters to split data into chunks and process them one at a time.

3.4.1 Data re-packing

We first determine a desirable MAC bit-length configuration A_b, usually the smaller bit-length of input and weight tensor. We then split each data point and distribute them to an additional tensor dimension *bit channel* if the data bit-length is larger than A_b, X_b, W_b, O_b for input, weight, output respectively, note that O_b is dependent on the A'_b of the next layer. Say we are going to process a layer using 4-bit multiplication, A_b is chosen to be 4; in Figure 3.4, original 4-bit data stored in wasteful 16-bit is re-packed along the **channel** dimension into new channel size of $C/4$; Figure 3.5 re-packs 8-bit data with additional X_b dimension of size 2.The re-packed data is meant to be added up within itself, and the data with different *bit channel* bear different weighting of $2^{X_b * A_b}$. In the example, we need a arithmetic unit capable of summing up 4 4-bit multiplication within a 16-bit data, we call this ***subword accumulation*** arithmetic dataflow, in contrast to the *subword parallelism* dataflow where multiple output registers are needed, only one output register is

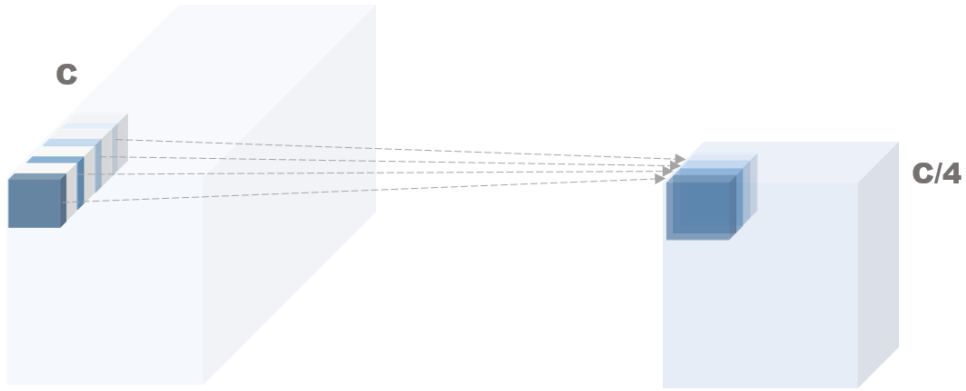


Figure 3.4: 4-bit data repacked to 16-bit compact data.

needed at a time. We will elaborate in chapter 4.

Having the data prepared through quantization and re-packing, we are ready to move on to the accelerator architecture designed for such data.

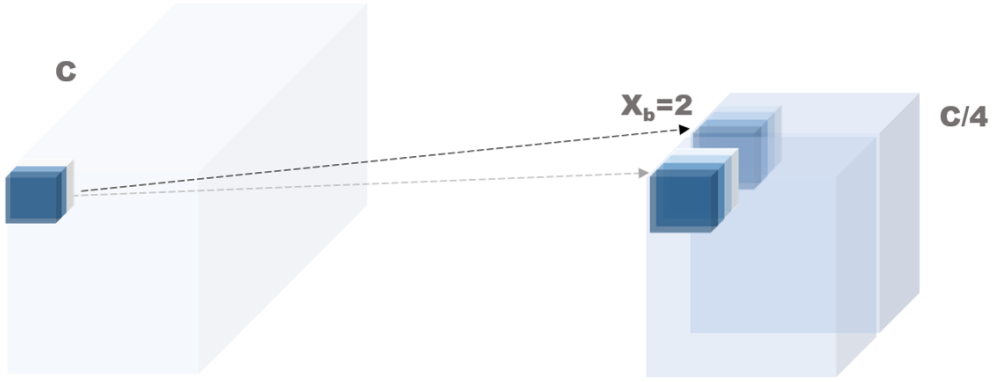


Figure 3.5: 8-bit data repacked to 16-bit compact data with additional bit channel data of size 2.

Table 3.3: Low bit arithmetic and shape parameters

Parameter	Description
X_b	Input bit channels
W_b	Weight bit channels
O_b	Output bit length
A_b	Arithmetic bit length
A'_b	Arithmetic bit length of the next layer

Chapter 4

Proposed Architecture

The architecture design is targeted to capture as much data reuse opportunity as possible, a three-level hierarchy on-chip buffer exploits almost every data reuseability in a regular convolution layer in the mean time saving power by logistically decreasing buffer size: data stays longer in the lower and smaller hierarchy buffer. As parallelism goes up, dispatching memory to each PE becomes a huge burden to the control logic; with data rearrangement, we can efficiently transfer data between buffer and buffer, or buffer and PE with shifter instead of MUX, saving logic and potentially power. We propose three micro-architecture dedicated to low-bit multiplication adder tree, able to operate on 1,2,4,8 bits signed and unsigned data, and an additional XNOR functionality. Finally a 5-stage pipeline PE structure is proposed to process *output row stationary* convolution. Power consumption is usually the bigger concern before performance when it comes to deploying DNN onto mobile devices. It has been shown that the main source of energy consumption comes from memory access from buffers and off-chip memory [11][12]. In comparison to memory access from register file or simple arithmetic operations, memory access to several dozen KB of SRAM uses an order of magnitude more energy, and yet off-chip memory access from DRAM takes as much as three orders of magnitude more energy. This sets our goal straight to have the data stay at lower energy consumption unit as longer as possible.

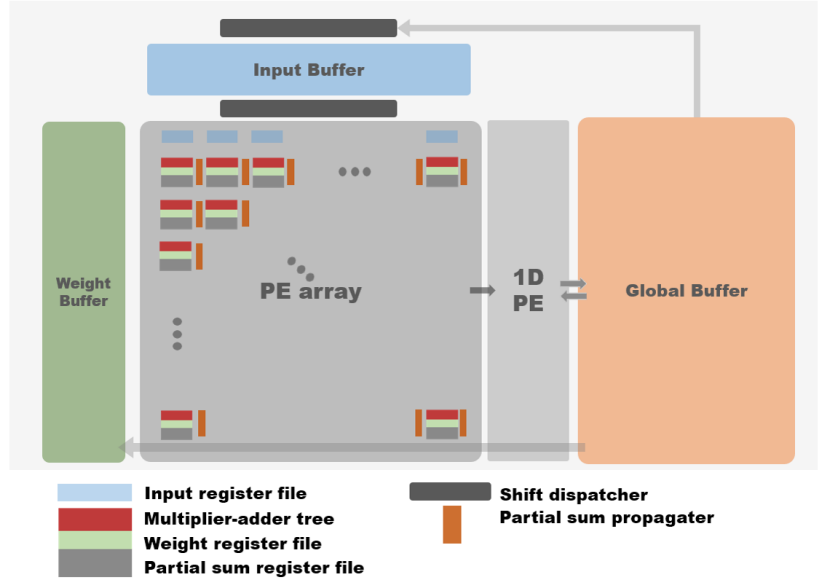


Figure 4.1: System architecture.

4.1 System Overview

Figure 4.1 shows the top-level architecture and memory hierarchy of the proposed system. The system consists of a 16x16 PE array, 100KB global single-port SRAM buffer, a column of size 16 of single-port 8KB weight buffer, a row of size 16 of single-port 16KB input buffer, a 1D PE taking care of post-processing. With careful data axis re-alignment, module-to-module data requires at most a shift from bank to bank without costly MUX. A PE column processes in lock-step, passing finished partial sum to neighbor PE column on their right until the edge of PE array, getting rid of complex routing logic needs for collecting data from PE array to global buffer. Input and Weight buffer broadcast their data to PE array along column and row respectively, similar to [6][7], once again requires no costly *network-on-chip* routing data as [12] does. Each PE takes rows of data each time, producing one row of output partial sums, capturing every convolutional reuse opportunities in its register files and partial sum register; inherited from [12] naming convention, we call this *output row stationary*.

4.1.1 Dataflow

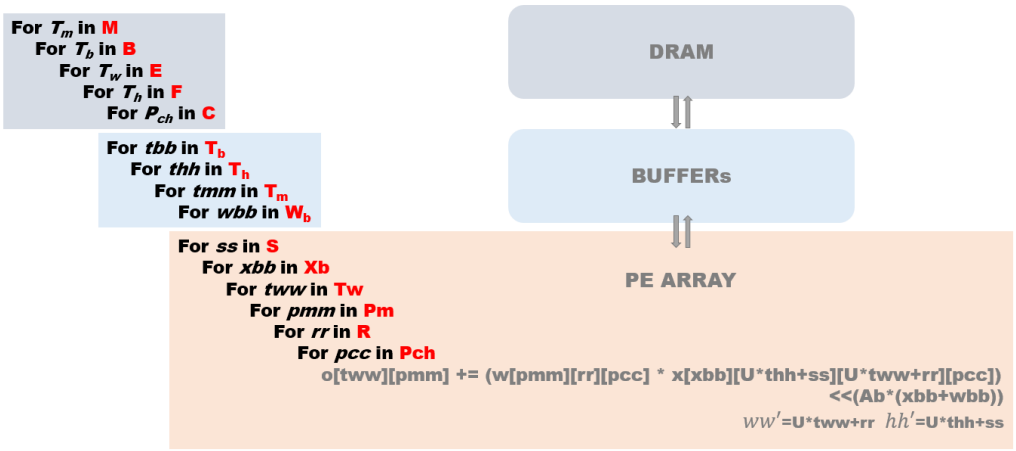


Figure 4.2: An overview of the proposed dataflow.

Figure 4.2 shows the outlook on the dataflow of one layer of CNN on our architecture, definitions of some looping factors can be found in the following sections. The figure gives an idea of data reuse opportunity spread through three level of memory hierarchy of off-chip DRAM, on-chip SRAM buffer, and register files inside each PE. Each PE is in charge of one output partial sum row and computes 1D convolution over a number of dimension; as soon as the computation is done, the output row written to the global buffer is never accessed again before switching to the next input channel tile, reusing data at its best in other word. The input and weight buffer reuse the data *spatially row-wise*, each data in the both buffer will be accessed at least S times. The trickier part lies in the input row data reuse, we spread input rows across input buffer banks based on stride U, with a simple shift, each column of PE can gain access to the data every cycle, granting the best bandwidth input buffer can offer, at the same time reuse data along S dimension. Further but relatively rare reuses include input data along W_b dimension, weight data along T_h or/and T_b dimensions.

The permutation of the for loops has a heavy impact on the data re-usability out of PE loops. Since the buffer size is limited, the deeper nested the loop is, the

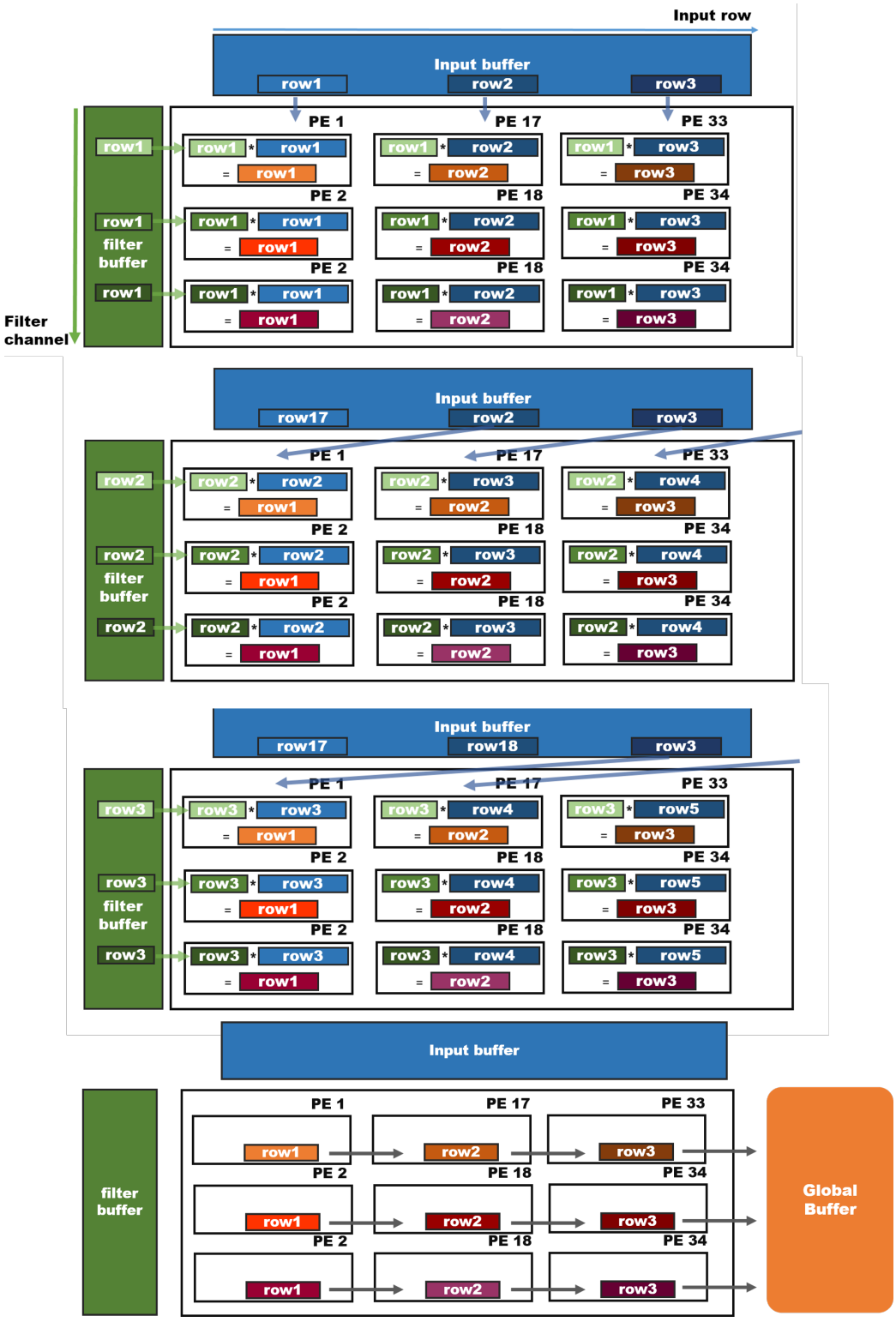


Figure 4.3: PE performing 1D convolution on a partial sum row.

frequent the data has to be thrashed, once this happens, the re-usability in the upper loops is forfeited. Therefore for loops in blue blocks in Figure 4.2, we arrange looping dimension as deeper as more likely the index to be 1.

Figure 4.3 demostrates the ***output row stationary*** dataflow where each PE keeps an partial sum row stationary, taking rows of input and weight from buffers and perform 1D convolution, and output the partial sum through the partial sum propagator out of the PE array to global buffer or off-chip.

4.1.2 Data tiling

Data tiling is widely used in the computation of large matrix-matrix multiplication, and is necessary for NN application. Figure 4.4 gives an overview of the four tiling parameters we use. The gray dotted line enclosed cubes are the data we would pass to the processor at each iteration. P_{ch}, P_m are the channel size and filter size processed by PE at a time.

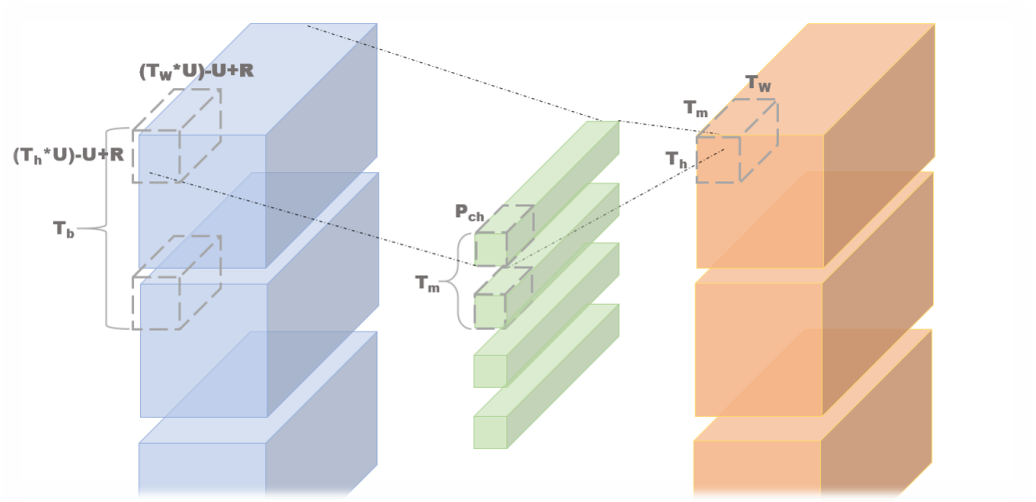


Figure 4.4: Tiling parameters on tensors.

Table 4.1: Tiling shape parameters

Parameter	Description
T_m	filters
T_w	feature map Width tile
T_h	feature map Height tile
T_b	Batch tile

4.1.3 Data re-alignment and buffer hierarchy

The data is specially arranged in the following manners Table 4.2 in global buffer. The global buffer is a 32-banks single-port SRAMs arranged in 2 groups, providing maximum 32x32b bandwidth per cycle. This packs the data in a way the data can be access from all the banks without bank conflict, granting maximum bandwidth. The tensors are viewed as collections of *row tensors*, couple with indexes to the rows for data re-alignment purpose. First the tensors are re-arranged as multi-dimensional array of *row tensor* and *row index tensor* parts, the group and bank are chosen based on the flattened tensor index, and compact row tensors accordingly in a buffer bank.

Figure 4.6 shows how the filter tile T_m is split-ed based on P_m, in order for the output quantization module packs contagious channels together, therefore saving off-chip memory access. This is the main contribution of this work, with the constraint of multiples of 16 filters computed by 16 PE rows in lock-step, so the output can be quantized and re-packed online. Input and Weight bank counts matches the PE array dimensions, so the data is accessed from each banks in one cycle, and broadcast-ed to PE columns or rows.

Figure 4.5 shows how data would be arranged in input and weight buffers. For input data specially, they are spread-ed across banks based on stride U, so the PE array can have required data with only a single shift from the output of input buffer. For example for 11x11 convolution with stride 4, the first output row takes **Row0** to **Row10**, the second output row takes **Row4** to **Row14**, at the first row

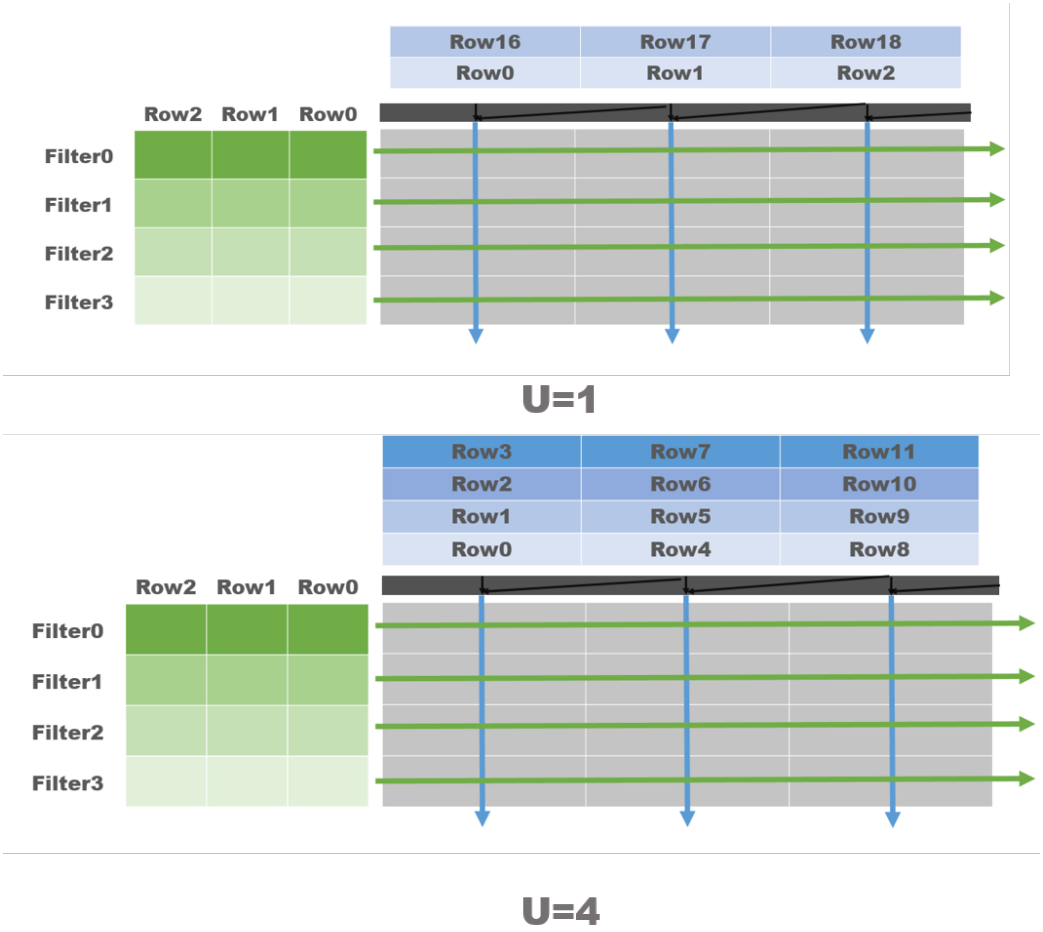


Figure 4.5: Input and Weight buffer data arrangement and their relation with stride U.

iteration, **Row0 Row4** can be accessed simultaneously, at the last row iteration, **Row10 Row14** can be accessed simultaneously with a left-shift of 2, operated by the shifter dispatcher in black box between PE array and input buffer.

Table 4.3 shows the specifications of the three data buffers. The parameters are mainly chosen based on worst case CNN shape parameter. The constrains are given by Table 4.4. Input buffer size is chosen based on **AlexNet** *conv1* layer with $W' = 63, P_{ch} = 1, X_b = 2, U = 4$. Weight buffer size is chosen based on **AlexNet** *conv1* layer with $R = S = 11, P_m = 2, P_{ch} = 1$. Global buffer size is chosen with

Table 4.2: Global buffer address space mapping

Row tensor	Row index tensor	bank	group
$\boldsymbol{I_r}[X_b][W'][P_{ch}]$	$\boldsymbol{I_{idx}}[T_b][H']$	$\lfloor \boldsymbol{I_{idx}.flatten/U} \rfloor \%16$	$\boldsymbol{I_{row}.flatten}\%2$
$\boldsymbol{W_r}[S][R][P_m][P_{ch}]$	$\boldsymbol{W_{idx}}[\lceil T_m/P_m \rceil][W_b]$	$\boldsymbol{tmm}\%16$	$\boldsymbol{W_{row}.flatten}\%2$
$\boldsymbol{O_r}[T_w][P_m]$	$\boldsymbol{O_{idx}}[\lceil T_m/P_m \rceil][T_b][T_h]$	$\boldsymbol{tmm}\%16$	$\boldsymbol{O_{row}.flatten}\%2$

$H' = (T_h * U) - U + R \quad W' = (T_w * U) - U + R$

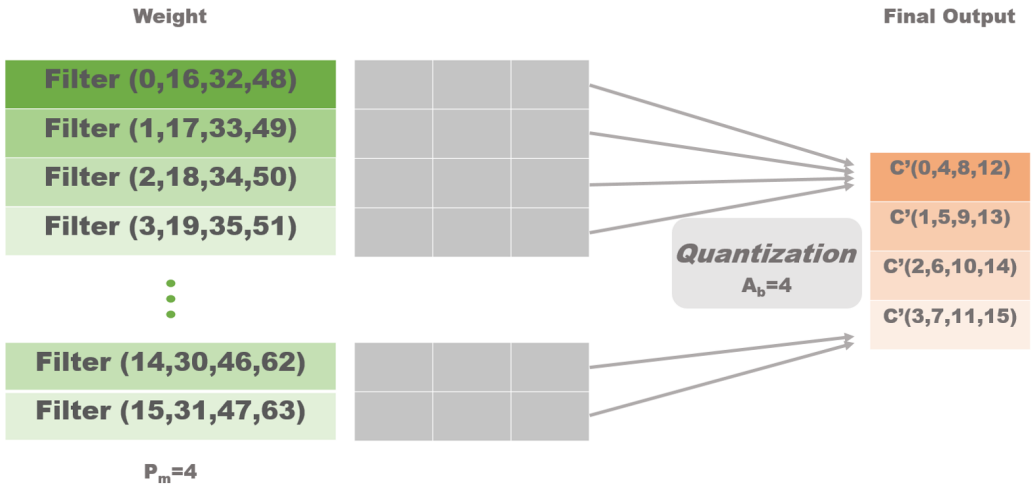


Figure 4.6: Filter index re-arranged based on P_m, so the output can be quantized and re-packed sequentially channel-wise.

based on **AlexNet** *conv1* layer with a set of processing parameters to be of 100KB, the configuration will be provided in chapter 5.

4.2 Architecture

4.2.1 PE processing pipeline

PE consist of a 3-stage processing pipeline, three register-files holding input, weight and partial sum, and a partial sum propagate pipeline. The first stage is the *Fetch stage*, its serves solely as a delay unit for register file data read. The second stage is the main contribution of this work, the *Multiplication stage* consisting of

Table 4.3: Buffer specifications

Buffer	banks	size	width	total size
Input	16	512	16	16KB
Weight	16	256	16	8KB
Global	32	800	32	100KB

Table 4.4: Buffer constraints

Buffer	constraints
Input	$W' * P_{ch} * X_b * U \leq 512$
Weight	$R * S * P_m \leq 256$
Global	$Input + Weight + Partial\ sum\ size \leq 100KB$
	$Input\ size = H' * W' * P_{ch} * X_b * T_b$
	$Weight\ size = T_m * S * R * W_b * P_{ch}$
	$Partial\ sum\ size = T_b * T_m * T_h * T_w * psum\ width$

psum width is either 16/32 bits

multiplier-adder trees that multiplies and add two 16-bit data subword-ly. The third stage *summation stage* is made of a 32-18 bit signed adder and a shifter. The adder can be configured to be a 16-16 bit signed adder for 16-bit partial sum. The shifter left shift the data $A_b * (wbb + xbb)$ bits, matching the data weighting according to the current bit-channels. The summation operation saturates any overflow or underflow data. The partial sum stays in the partial sum register in this stage for $P_{ch} * R$ cycles, and then written to *PPAD*, the partial sum register file.

The final stage *output propagate path stage* works as a FIFO, it takes partial sum from neighbor PE on its left or read from the PPAD, and passes the partial sum down the propagate path to its neighbor PE on the right. The stage chooses its partial sum source in a round-robin manner, if both partial sums from left PE and PPAD are available, it takes the source other than the last source processed. The propagate path throughput is 16x32bit, it has to sends out *Active PE row* $* P_m * T_w$ number of data before the new partial sum row being processed overlaps the PPAD address

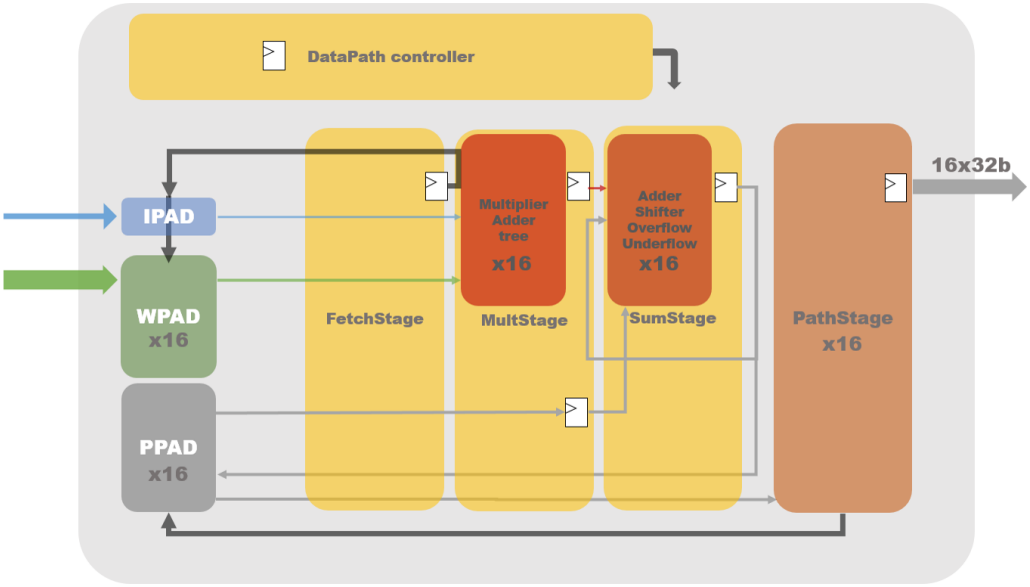


Figure 4.7: PE column pipeline.

without any latency. The overlapping condition is $16{*}P_m{*}T_w \geq R{*}P_{ch}{*}2{*}T_w{*}P_m$, which means that the PE can't generates two rows of partial sum with the cycles it takes to transfer 16 rows of partial sum out of the PE array without stalling the PE processing, 16 is the worst case where every PE columns are active. In practice, even the unwanted latency presents, it doesn't deter the entire processing much as a partial sum are sent out every $R * S * P_{ch} * T_w * P_m * X_b$ cycles, the latency would always be negligible. As we can see the condition is always met for fixed PE array dimension, it limits the scalability of the architecture, the column number can't be too big for global buffer bandwidth to bear.

Now to the data *scratch pad*. Every PE is equipped with one of each input, weight and partial sum scratch pad. The scratch pads are register files that hold data for the purpose of convolutional data reuse. The scratch pad specifications and constraints are given by Table 4.5 and Table 4.7. The input and weight pad hold a window of current 1D convolution window, the partial sum pad instead holds P_m rows of partial sum. Note that the partial sum PAD is a 32x32b register file, capable of handling both 16-bit and 32-bit partial sum data. Practically configuration larger than 8-bit and 4-bit input and weight requires 32-bit partial sum to prevent constant

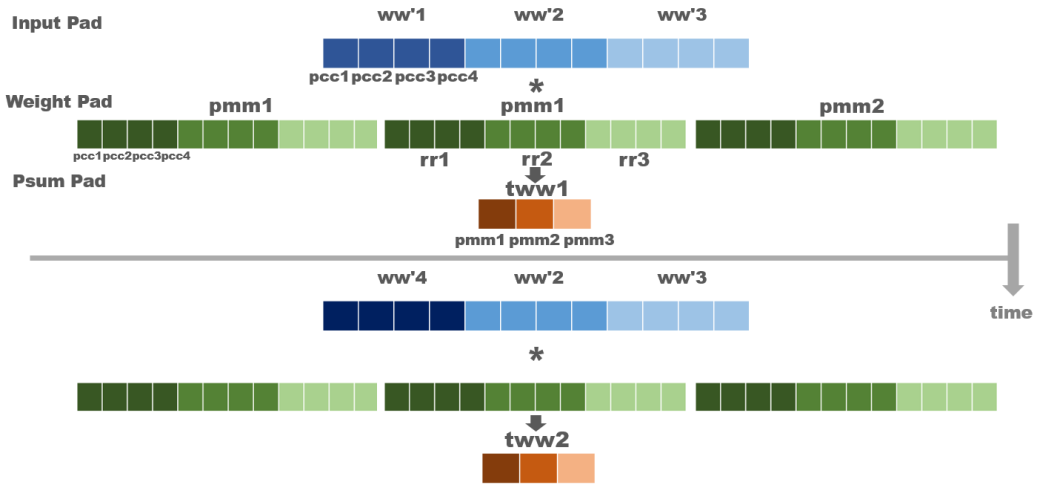


Figure 4.8: An example of 1D convolution with $P_{ch} = 4, P_m = 3$, at finishing first partial sum pixel tww1, input pixel ww'1 is replaced in input pad with ww'4, and proceed to partial sum pixel tww2.

overflow or underflow, yet a 4-bit to 4-bit setting, which consists most portion of our best **AlexNet** model, 16-bit partial sum is enough. With knowledge of scratch pads, we provide an example in Figure 4.8 and Figure 4.9 demonstrating the data storage in pads and 1D convolution dataflow in terms of looping indexes.

To be able to quantize and re-pack output data as soon as the partial sum is finished, a column of PEs need to process in lock-step. We then group a column of PEs as an unit, this gives us several opportunities for hardware sharing. The main controller is shared across a PE column, a main controller in a PE actually takes up about 5% of area, if not shared, approximately an additional 54k gates area is needed for each column. Filter reuse along row in a PE column is a significant data reuse-ability, the input data from input pad is shared for every filters from different PE rows. We then put only one input pad in a PE column accordingly, this saves 360B of register file storage space in a PE column. Since PE array is of large 16-column, the mentioned resource sharing contributes hugely.

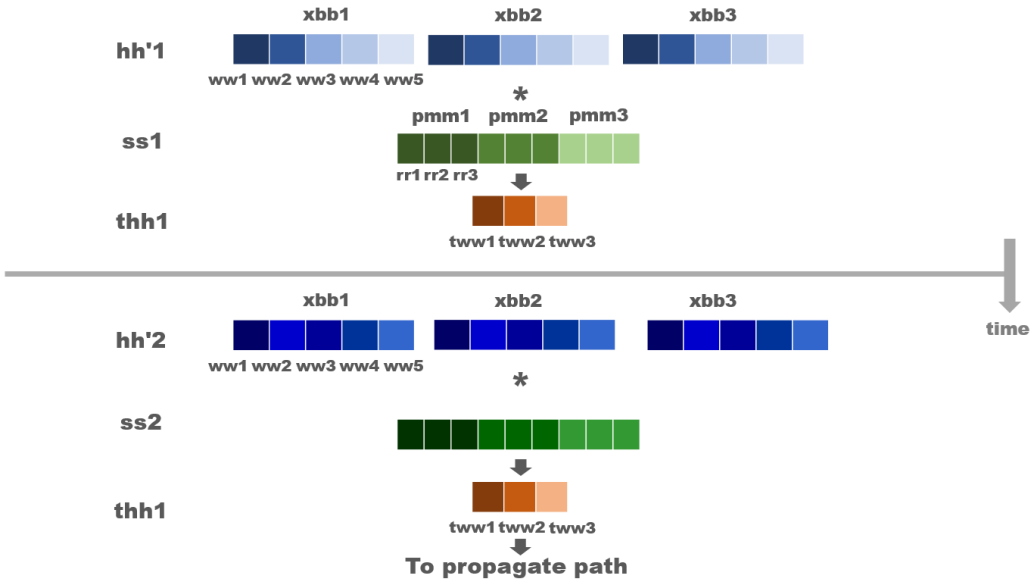


Figure 4.9: An example of 1D convolution with $T_w = 3, X_b = 3, S = 2$, at finishing first convolution row after 3 input bit channels, we proceed to second row convolution ss2, and then finish the output row thh1, sending partial sum out of PE.

Table 4.5: PAD specifications

PAD	size	width	total size (entire chip)
Input	12	16b	384B
Weight	48	16b	24KB
Partial sum	32	32b	32KB

4.2.2 Re-configurable arithmetic logic unit

First we review the low-precision operation details. The input and weight data is re-packed along channel dimension based on arithmetic bit length A_b into a data of 16-bit word. The channel dimension in CNN is meant to be accumulated together. Figure 4.10 shows a 4x 4-bit multiplication and addition is needed for such data. Similarly, we need 16x MAC for 1-bit data, 8x for 2-bit, 2x for 8-bit. We call such basic operator **MAT** as in *multiplier adder tree*, as it performs $16/A_b$ multiplications and sums them up. The multiplication needs to support either signed or unsigned mode. We also include *XNOR* mode feature binarized layer as

Table 4.6: PAD constraints

PAD	constraints
Input	$R * P_{ch} \leq 12$
Weight	$R * P_m * P_{ch} \leq 48$
Partial sum	$T_w * P_m * psum\ width \leq 64$

psum width is either 1/2 word, a word is 16-bit

in [3], where each binary data 1 mapped to 1, 0 to -1.

We propose three micro-architecture for such operations. The operators must

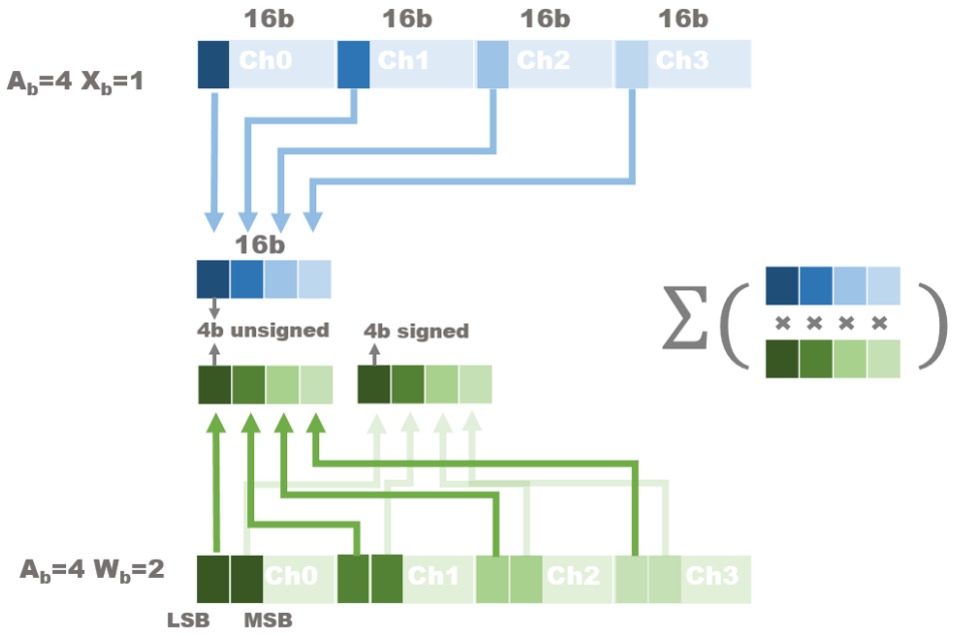


Figure 4.10: 4-bit input and 8-bit weight re-packed using A_b=4.

support signed and unsigned 16x1b, 8x2b, 4x4b, 2x8b MAT and XNOR operation. The first one is 8-bit *simple multiplication* MAT. The module takes an un-optimized multiplier as a base, masked out unused logics based on configuration, and add the 8 partial-product together for the final product. For XNOR mode, the partial product is replaced with XNOR gate rather than AND gate, then simply takes the 1-bit summation result P, which is essentially a bit-counting operation, and outputs

$2 * P - 16$. Re-configurability on 5 modes with support of signed and unsigned data makes such design induces at least a 2-to-1 MUX on each bit of partial-product, as to be zero-gated, sign extended or the original partial-product. The routing cost is not to be overlooked, not to mention the long critical path of 8+8 ripple carry adder. The advantage of this design lies in no MUX for the final-product and the inputs.

The second one *booth MAT* is based on booth-algorithm multiplier. A typical

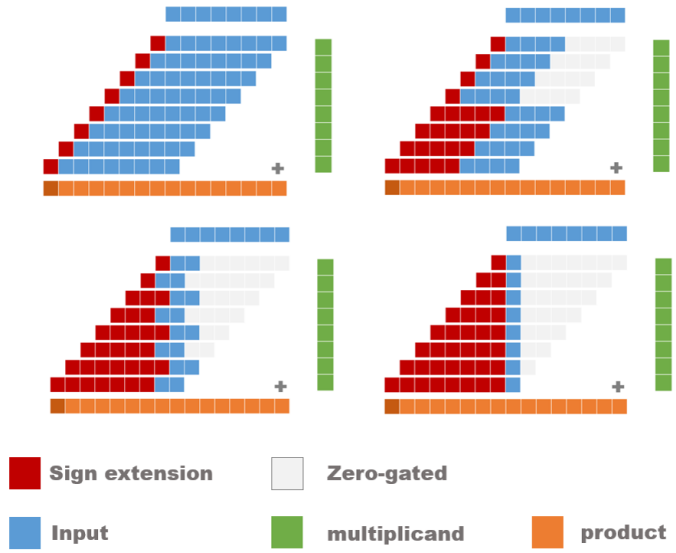


Figure 4.11: 8,4,2,1-bit mode of simple MAT from upper-left to lower-right.

radix-4 signed booth multiplier reduced the partial-product from 8 down to 4. Radix-4 setting straight up implies no partial-product and critical path saving for 2-bit and 1-bit operations, since it produced a partial-product per 3-bit multiplicand (the weight or the green bits), larger than what 2-bit and 1-bit need. We have to add additional partial product for unsigned setting and 2-bit, 1-bit mode. The basic working principle is provided in Figure 4.12. To perform 2-bit or 1-bit MAT on a 8-bit booth multiplier, additional partial products and new multiplicand zero-gated rules are added. The reasoning can be developed from piecing together wanted 2-bit partial product, as in Figure 4.13, we need the two partial product shown in the right portion of the figure, so we need the two zero-gated 3-bit multiplicand and an additional partial product based on the gated multiplicand. In turn, the routing

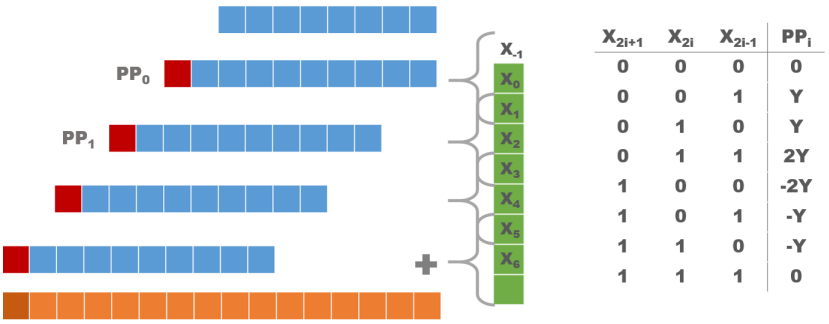


Figure 4.12: Partial product relation to 3 multiplicand bits in a group in a 8-bit signed booth multiplier.

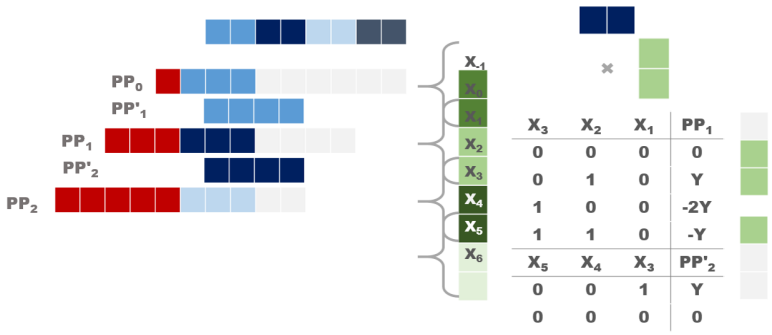


Figure 4.13: To perform 2-bit MAT on a 8-bit booth multiplier, the multiplicand will be properly zero-gated, the partial product has to carefully chosen, additional partial product needs to be added.

and MUX-ing in this design is unbearably complicated, so as the synthesis result show. The additional partial products needed is shown in Figure 4.14.

The last design *Mux MAT* is the simplest yet the most efficient in every aspects. It takes in 16-bit input and weight, broadcasts both data to 4 set of multipliers and adder trees units, however zero-gated input port to the idle units. The multipliers and adder trees bits are chosen in order to support both signed and unsigned data by sign extending the product, that is 1,5,9,17-bit for 1,2,4,8 multiplication respectively. The output bits are chosen for the maximum required bit-length of the adder tree output. XNOR mode takes the XNORed gates bit-counting results,

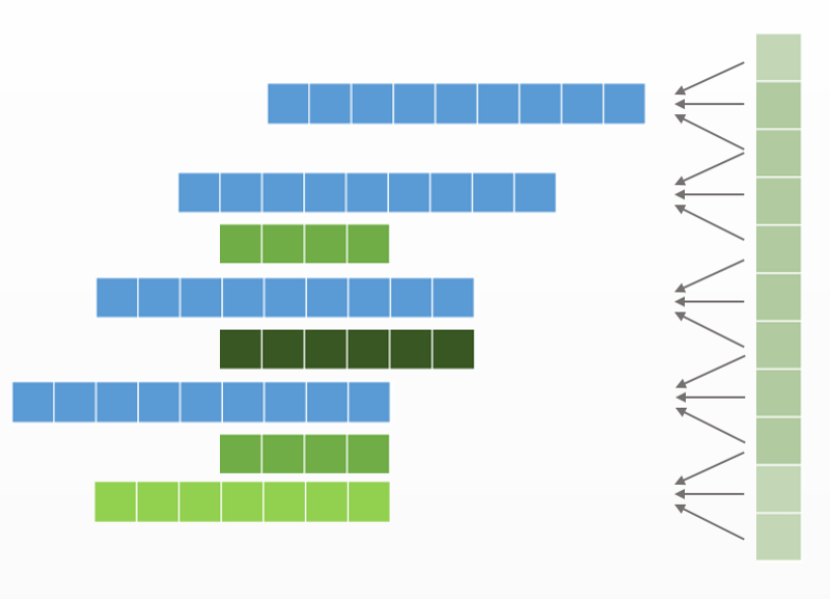


Figure 4.14: The additional partial product just for 2-bit and 1-bit configuration.

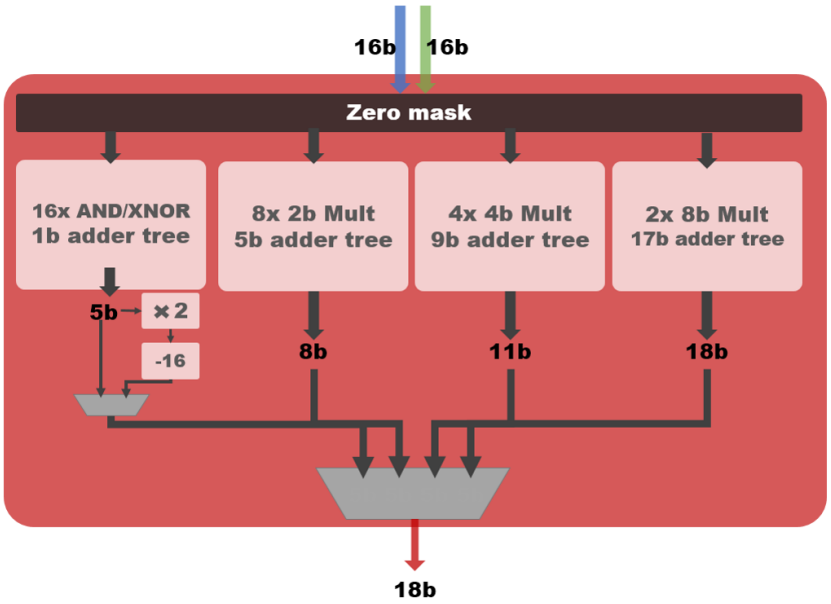


Figure 4.15: MATmux chooses the configured product as the output, masked out idle adder trees and multipliers.

multiplies it by 2 and subtract 16.

Synthesis results concludes for using MUX MAT design. For the time being, we focus on the slack where it determines the critical path of the system, it plays the

Table 4.7: Synthesis for MAT designs

Design	cell area (TSMC 90nm)	slack (400MHz)
simple multiplication	11815* [1]	0
booth	9618*	0
MUX	13659	0.30
16b multiplication (32b output)	11486	0**
16b multiplication (16b output)	4274	0**[2]

most important role in low-power consumption goal. Simple mult design and booth both haven't met our desired constraint, so we did not do thorough verification on them, the synthesis area would possibly go up.

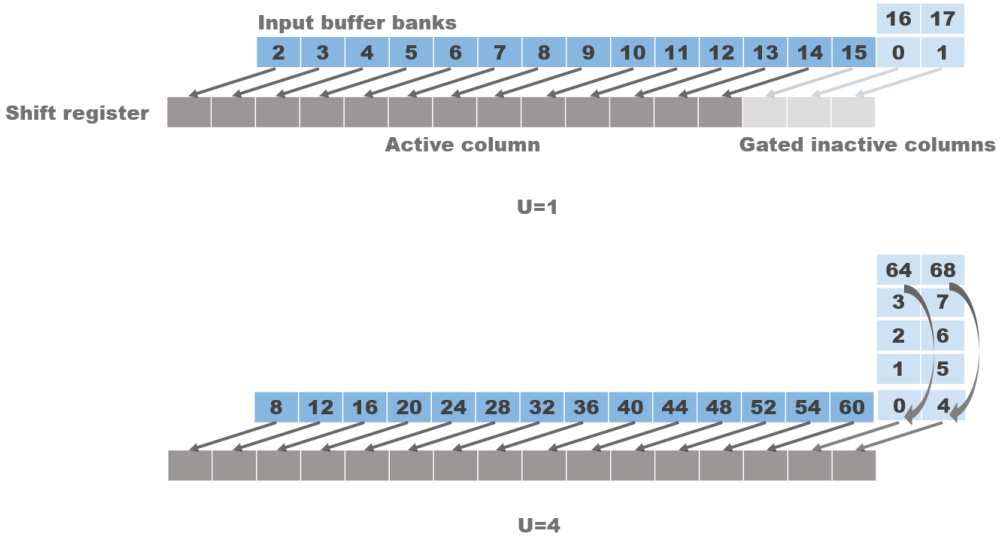


Figure 4.16: Following previous example, each PE column access needed data simultaneously, the bandwidth is not wasted.

[1] *correctness not checked

[2] ** typically 16-bit multiplication won't fit in a 2.5ns cycle time even on 65nm process

4.2.3 Shift dispatcher

Shift dispatcher between buffers and PE array are just a circular shift-register. As the input is re-arranged based on stride U either off-line (first layer) or naturally aligned by the processing of the system at the output, every PE columns can access required data from each input banks at the same cycle. The registers are gated if corresponding PE column is inactive. Figure 4.16 demonstrate a $U = 1, S = 3, ss = 3$ and a $U = 4, S = 11, ss = 9$ example where left shift 2 is needed for both.

4.2.4 Quantization

We use round to the nearest quantization scheme. After the final scaling and shifting, we save two digit behind the decimal point, in other word we right shift $\gamma' = gamma - 2$ 2 less digit than the real shifting factor. Now for positive number, the result add 1 if the first digit behind decimal point is 1, otherwise get rid all the decimals. For negative number, the result add 1 if the two digits behind decimal point is 11, otherwise get rid of all the decimals. For example a positive number 01000.1_b is rounded to 01001, that is 8.5 rounded to 9; a negative number 10111.11_b is rounded to 11000, that is -8.25 rounded to -8.

Chapter 5

Results

5.1 Quantization error minimization training

Our quantization error minimization training on AlexNet achieves 54% accuracy with the following setting in Table 5.1. Note that we apply quantization on the first layer as well, and some of the model shape doesn't follow a typical AlexNet model. The input bit-length stays 8-bit as original. Table 5.2 shows the classification accuracy of several works on ImageNet using AlexNet, note that the first layer of XNOR-net is not quantized, and possibly pose a large computational overhead over quantized one.

Table 5.1: AlexNet 4-bit quantization

layer	conv1	conv2	conv3	conv4	conv5	Fc1	Fc2	Fc3
Weight	4	4	4	4	4	2	2	32
Activation	4	4	4	4	4	4	4	32
Output channels	64	256	256	256	256	4096	4096	1000

Table 5.2: Classification Accuracy on ImageNet, AlexNet

	XNOR-Net[3]	TensorRT[4]	OUR
Bit-length	1	8	4
Accuracy	44.2%	57%	54%

5.2 Implementation results

We synthesize and evaluate our design using TSMC 90nm technology. The PE array and Buffer specification is provided in Table 5.3, the best synthesized cell area is 15698896 cell area, approximately 15.6 mm^2. We co-simulate the design using **Nicotb**[23] and Cadence nc-verilog version 15.20-s039; we synthesize the design with Synopsys design compiler version 0-2018.06 and finally simulate power consumption with Synopsys PrimeTime version M-2017.06-SP2. Table 5.4 lists several highly related hardware implementation works in comparison, note that Vgg16 benchmark setting is not yet trained on 4-bit arithmetic setting, yet we are highly confident that there is room for quantization for such redundant network.

Table 5.3: System specification

Technology	TSMC 90nm GUTM Arm f1.0
Gate count (logic only)	1340.7k (NAND2)
Area	15.6mm^2
PE dimension	16x16
On-chip SRAM	124KB
Register file	56K+384B
Clock rate	200-400MHz
Peak throughput	51.2N-102.4N GOPs, N=16,8,4,2
Arithmetic precision	re-configurable 1,2,4,8-bit fixed-point
Average full power	672mW @200MHz
Power efficiency	76.19N-152.38N GOPs/W, N=16,8,4,2

Table 5.4: System specification comparison.

Work	JSSC'17[12]	VLSI'16[6]	ISSCC'17[7]	This work N=16,8,4,2
Technology	65nm LP CMOS	40 LP CMOS	28nm UTBB FD-SOI	90nm CMOS
Tape-out	V	V	V	pre-sim
Frequency	200	200	200	400
Supply voltage	1	1.1	1	1
Peak GOPs	67	102	nx102 n=1,2,4	Nx152.38
Area (mm^2)	12.25	2.4	1.87	15.6
# of MACs	168	256	nx256 n=1,2,4	Nx256
Gate Counts	1.176M	1.6M	1.95M	1.34M
On-chip memory (KB)	184.5	144	144	180
Precision	16-bit fixed	Dynamic 1-16	Dynamic n x 1-16/n	1,2,4,8
AlexNet benchmark	278mW @34.7fps	76mW @ 47fps	44mW @47fps	506mW avg @206.9fps,4b
Vgg16 benchmark	236mW @0.7fps	-	26mW @1.67fps	586mW @ 6.86fps,4b,N/A acc

5.2.1 Area and power

Figure 5.1 shows the area breakdown chart for this work, PE logic takes up 25% of area in contrast to 75% of memory, which actually accounts for a large portion in comparison to related work, as multipliers in [12] takes up 7.4% of area.

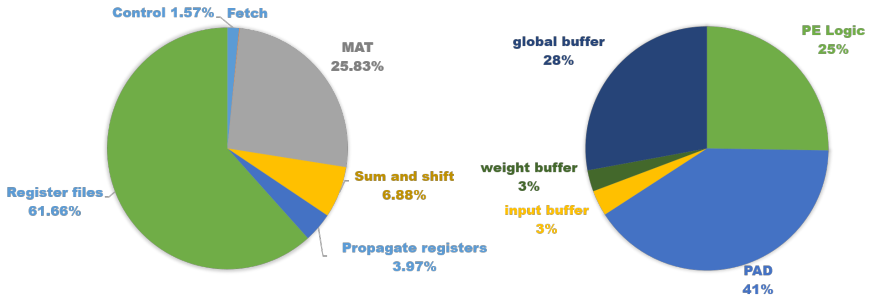


Figure 5.1: Area breakdown of the system.

Figure 5.2 shows the buffer hierarchy significance, removing input and weight buffer introduces a 1.14x power consumption overhead on representative AlexNet conv5 layer. The additional buffer takes up about 20% more area as a trade-off. Figure 5.3 shows the power breakdown of the system. The PE array consumes most portion of the power, indicating workloads falls mainly on the most power efficient part of the system. The PAD access, MAT, sum and shift draws close

Table 5.5: PE column area gate counts

control	Fetch	MAT	Sum and shift	Propagator	IPAD	PPAD	WPAD	total
3670	200	60503	16122	9295	2756	74856	66802	234207

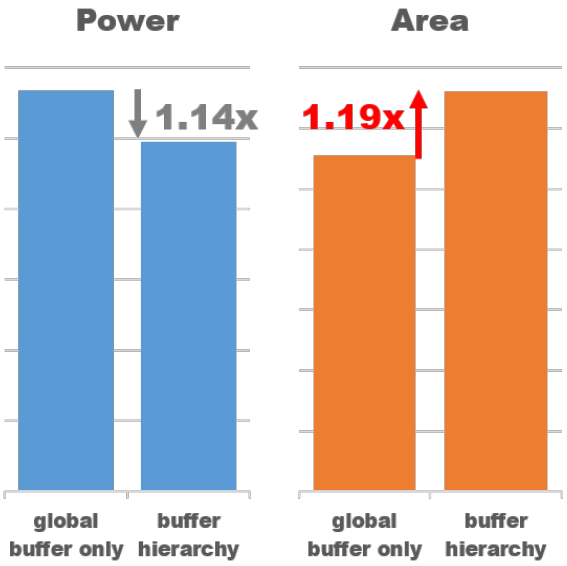


Figure 5.2: Buffer hierarchy area power trade-off.

amount of power. Table 5.6 shows a 4-bit layer power consumption simulated at 200MHz, 1V. Figure 5.4 shows the power consumption of different bit mode setting on MAT, indicating that the low-precision operations contributes mainly on memory access saving, the system benefits from arithmetic operation lightly.

Table 5.6: System average power (mW) consumption at 200MHz

DPC	FS	MS	SS	PS	PAD	buffer hierarchy	total
4.16	1.44	219	122.	7.68	218	79.1	652.5

5.2.2 Experiments

We lists the configuration and setting used on various models, and the performance reports. All the experiments set the batch size B to 4, processes at 200MHz, 1V.

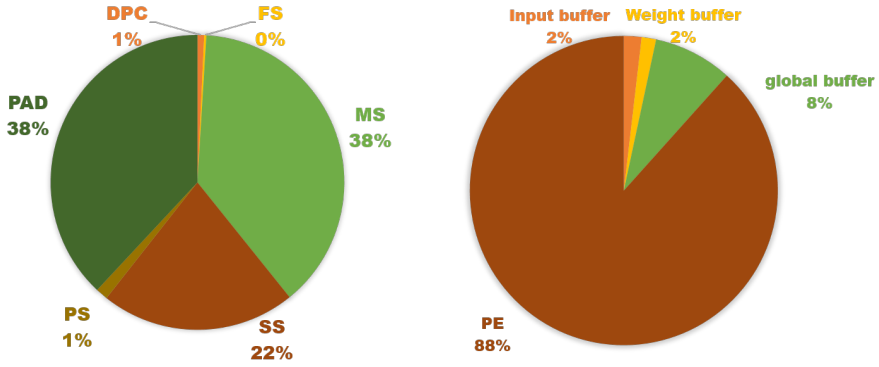


Figure 5.3: Average power breakdown of the system.

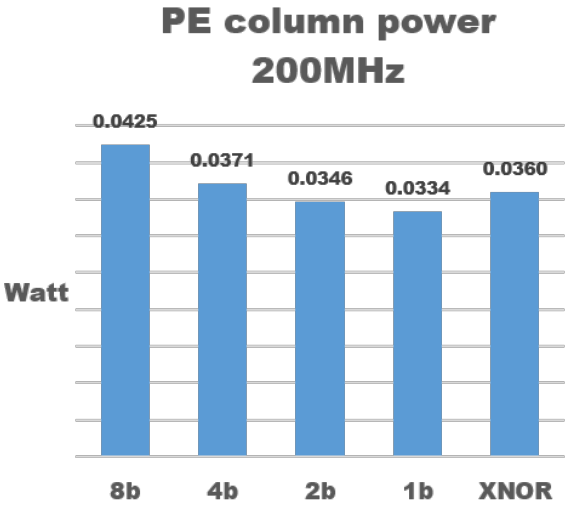


Figure 5.4: Arithmetic unit configuration power.

Table 5.7: Performance summary Batch=4

	FPS	processing time (s)	off-chip access(MB)	average power (mW)	Accuracy(%)
AlexNet	206.9	0.019	5.84	573	54
Vgg16	6.86	0.58	210.01	664	N/A
Alex XNOR	491.2	0.008	2.99	581	44
TensorRT Alex 8-bit	116.5	0.034	13.95	580	57

Note the $psum\ word$ parameter is the partial sum word length storage on-chip, a word is of 16-bit, we use 2 word 32-bit for partial sum when overflow takes place frequently. Our hardware utilization rate is easily determined, which is usually T_h,

the active PE column counts as well as the output rows processed in a PE row. If T_h is larger than the PE column number 16, we allocate as much resource, as in PE columns as possible, and gated idle PE columns at the last height tile. For the most part, our hardware utilization rate on ImageNet, where the input spatial size is of 224x224, is usually $13/16(81.25\%)$. The utilization rate can be easily scaled up if we fit the design to a particular dataset spatial dimension, however we choose the PE shape for general purpose, and multiple of 16 is usually suitable for hardware designs. From various experiments we can see the power of quantization as low as to 4-bit, it boosts the processing time and reduce off-chip memory access even without coding or compression at the edge of the system. More importantly, the design is highly flexible and fit for various bit-length and model shapes.

Table 5.8: AlexNet configurations

	T_w	T_h	H'	W'	P_{ch}	X_b	W_b	A_b	O_b	P_m	R	S	T_m	T_b	C	E	F	M	psum word
alex conv1	14	14	63	63	1	2	1	4	4	2	11	11	64	2	3	55	55	64	2
alex conv2	27	27	31	31	2	1	1	4	4	2	5	5	64	1	64	27	27	256	1
alex conv3,4,5	13	13	15	15	4	1	1	4	4	4	3	3	64	4	256	13	13	256	1
old alex conv3	13	13	15	15	4	1	1	4	4	4	3	3	64	4	256	13	13	384	1
old alex conv4	13	13	15	15	4	1	1	4	4	4	3	3	64	4	192	13	13	384	1
old alex conv5	13	13	15	15	4	1	1	4	4	4	3	3	64	4	192	13	13	256	1

Table 5.9: Vgg-16 configurations

	T_w	T_h	H'	W'	P_{ch}	X_b	W_b	A_b	O_b	P_m	R	S	T_m	T_b	C	E	F	M	psum word
vgg1	16	16	18	18	4	2	1	4	4	2	3	3	64	2	3	224	224	64	2
vgg1-2	16	16	18	18	4	1	1	4	4	4	3	3	64	4	64	224	224	64	1
vgg2-1	16	16	18	18	4	1	1	4	4	4	3	3	64	4	64	224	224	128	1
vgg2-2	16	16	18	18	4	1	1	4	4	4	3	3	64	4	128	112	112	128	1
vgg3-1	16	16	18	18	4	1	1	4	4	4	3	3	64	4	128	112	112	256	1
vgg3-2,3	16	16	18	18	4	1	1	4	4	4	3	3	64	4	256	56	56	256	1
vgg4-1	16	16	18	18	4	1	1	4	4	4	3	3	64	4	256	56	56	512	1
vgg4-2,3	16	16	18	18	4	1	1	4	4	4	3	3	64	4	512	28	28	512	1
vgg5-1	15	15	18	18	4	1	1	4	4	4	3	3	64	4	512	28	28	512	1
vgg5-2,3	14	14	18	18	4	1	1	4	4	4	3	3	64	4	512	14	14	512	1

Figure 5.5 studies the re-configurablility benefits and trade-offs on various bit-length setting. Most interesting part is the mixed precision flexibility of this

Table 5.10: XNOR AlexNet configurations

	T_w	T_h	H'	W'	P_{ch}	X_b	W_b	A_b	O_b	P_m	R	S	T_m	T_b	C	E	F	M	psum word
alexnor conv1	14	14	63	63	1	1	1	8	1	2	11	11	64	2	3	55	55	64	2
alexnor conv2	27	27	31	31	2	1	1	1	1	2	5	5	64	1	64	27	27	256	1
alexnor conv3	13	13	15	15	4	1	1	1	1	4	3	3	96	4	256	13	13	384	1
alexnor conv4	13	13	15	15	4	1	1	1	1	4	3	3	96	4	192	13	13	384	1
alexnor conv5	13	13	15	15	4	1	1	1	1	4	3	3	96	4	192	13	13	256	1

Table 5.11: TensorRT 8-bit AlexNet configurations

	T_w	T_h	H'	W'	P_{ch}	X_b	W_b	A_b	O_b	P_m	R	S	T_m	T_b	C	E	F	M	psum word
alex 8b conv1	14	14	63	63	1	1	1	8	8	2	11	11	64	2	3	55	55	64	2
alex 8b conv2	27	14	31	31	2	1	1	8	8	1	5	5	64	1	64	27	27	256	2
alex 8b conv3	13	13	15	15	4	1	1	8	8	2	3	3	64	4	256	13	13	384	2
alex 8b conv4	13	13	15	15	4	1	1	8	8	2	3	3	64	4	192	13	13	384	2
alex 8b conv5	13	13	15	15	4	1	1	8	8	2	3	3	64	4	192	13	13	256	2

Table 5.12: AlexNet results

	ibuf size (B)	wbuf size (B)	psum pad (2B)	gb size (B)	off-chip access (MB)	prcessing time (ms)
alex conv1	1008	484	56	97416	1.82	4.34
alex conv2	248	200	54	56900	1.61	6.91
alex conv3,4,5	120	288	52	55072	0.80	2.40
old alex conv5	120	288	52	55072	0.62	1.80
old alex conv4	120	288	52	55072	0.93	2.70
old alex conv3	120	288	52	55072	1.20	3.59
total(ours/typical)					5.84/6.19	18.44/19.34

work, as long as the training results permits, the potential presents. The figure compares the off-chip memory access and processing cycle counts on our AlexNet conv5 setting. As the data bit-length varies, the optimal arithmetic mode can be chosen based on the charts. Possibilities exist where the cost is close for two arithmetic configuration, the better choice lies in the trade-off between saving off-chip DRAM power consumption which is often 3-orders higher than a basic operation and saving processing cycle on a averagely 600mW system.

Table 5.13: Vgg results

	ibuf size (B)	wbuf size (B)	psum pad (2B)	gb size (B)	off-chip access (MB)	prcessing time (ms)
vgg1	288	288	64	80512	11.72	18.06
vgg1-2	144	288	64	80512	17.32	36.13
vgg2-1	144	288	64	80512	25.46	72.25
vgg2-2	144	288	64	80512	14.26	36.13
vgg3-1	144	288	64	80512	23.93	72.25
vgg3-2	144	288	64	80512	16.63	47.19
vgg3-3	144	288	64	80512	16.63	47.19
vgg4-1	144	288	64	80512	30.25	94.37
vgg4-2	144	288	64	80512	15.63	47.19
vgg4-3	144	288	64	80512	15.63	47.19
vgg5-1	144	288	60	72576	14.88	44.24
vgg5-2	144	288	56	65152	3.85	10.32
vgg5-3	144	288	56	65152	3.85	10.32
total					210.01	582.82

Table 5.14: Xnor AlexNet results

	ibuf size (B)	wbuf size (B)	psum pad (2B)	gb size (B)	off-chip access (MB)	prcessing time (ms)
alexnor conv1	504	484	56	81540	2.01	4.34
alexnor conv2	248	200	54	56900	0.40	1.73
alexnor conv3	120	288	52	79008	0.25	0.90
alexnor conv4	120	288	52	79008	0.19	0.67
alexnor conv5	120	288	52	79008	0.14	0.51
total					2.99	8.14

Table 5.15: TensorRT AlexNet results

	ibuf size (B)	wbuf size (B)	psum pad size (B)	gb size (B)	off-chip access (MB)	prcessing time (ms)
alexnet 8b conv1	504	484	56	81540	2.68	4.34
alexnet 8b conv2	248	100	54	58628	5.74	13.82
alexnet 8b conv3	120	288	52	98336	2.41	7.19
alexnet 8b conv4	120	288	52	98336	1.87	5.39
alexnet 8b conv5	120	288	52	98336	1.25	3.59
total					13.95	34.33

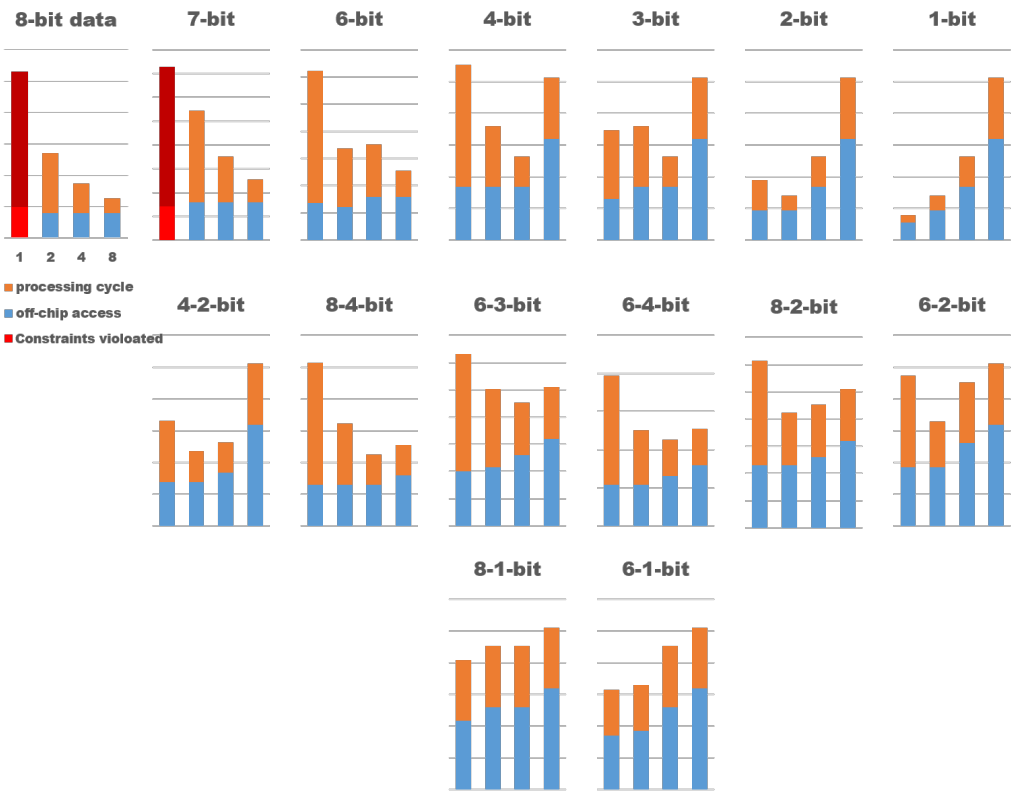


Figure 5.5: Various data bit-length to arithmetic bit-length off-chip access and processing time scaling.

Chapter 6

Conclusion

An architecture capable of performing 1,2,4,8-bit multiplication and add operation on CNN is presented. We set out with the feasibility of linear quantization scheme on CNN, the power consumption bottleneck for deploying CNN on mobile devices, to developing a promising training scheme for linear quantization threshold choosing and achieve 54% accuracy on a 4-bit AlexNet model on ImagenNet dataset, and finally design and implement a precision re-configurable hardware accelerator dedicated to low arithmetic precision CNN networks. The implementation uses 180KB on-chip memory and 1340K logic gates, comparable to existing designs. We synthesized and evaluated the system down to gate-level. The design uses a dataflow suitable for convolution layers with high data reuse rate, couple with a subword accumulation style operation so that the data shape fits onto the on-chip memory properly. We believe this work can benefit low-power mobile deep neural network deployment in practical use.

Reference

[1] E. H. Lee, D. Miyashita, E. Chai, B. Murmann, and S. S. Wong, "Lognet: Energy-efficient neural networks using logarithmic computation," in *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, March 2017, pp. 5900–5904. v, 6

[2] S. Han, H. Mao, and W. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," 10 2016. v, 2, 6

[3] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, "XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks," *arXiv e-prints*, p. arXiv:1603.05279, Mar 2016. v, 3, 6, 8, 18, 37, 44

[4] S. Migacz, "8-bit inference with tensorrt." [Online]. Available: http://on-demand.gputechconf.com/gtc/2017/presentation/s7310-8-bit-inference-with-tensorrt.pdf v, 8, 9, 44

[5] Y. Chen, T. Krishna, J. Emer, and V. Sze, "14.5 eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," in *2016 IEEE International Solid-State Circuits Conference (ISSCC)*, Jan 2016, pp. 262–263. [Online]. Available: http://people.csail.mit.edu/emer/slides/2016.02.isscc.eyeriss.slides.pdf v, 10

[6] B. Moons and M. Verhelst, "A 0.3-2.6 TOPS/W precision-scalable processor for real-time large-scale convnets," *CoRR*, vol. abs/1606.05094, 2016. [Online]. Available: http://arxiv.org/abs/1606.05094 v, 11, 26, 45

[7] B. Moons, R. Uytterhoeven, W. Dehaene, and M. Verhelst, "14.5 envision: A 0.26-to-10tops/w subword-parallel dynamic-voltage-accuracy-frequency-scalable convolutional neural network processor in 28nm fdsoi," in *2017 IEEE International Solid-State Circuits Conference (ISSCC)*, Feb 2017, pp. 246–247. v, 11, 26, 45

[8] H. Sharma, J. Park, N. Suda, L. Lai, B. Chau, J. K. Kim, V. Chandra, and H. Esmaeilzadeh, "Bit fusion: Bit-level dynamically composable architecture for accelerating deep neural networks," *CoRR*, vol. abs/1712.01507, 2017. [Online]. Available: http://arxiv.org/abs/1712.01507 v, 12

[9] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," *arXiv e-prints*, p. arXiv:1512.03385, Dec 2015. 1

[10] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems 25*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2012, pp. 1097–1105. [Online]. Available: http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf 1, 3

[11] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "EIE: Efficient Inference Engine on Compressed Deep Neural Network," *arXiv e-prints*, p. arXiv:1602.01528, Feb 2016. 1, 2, 25

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

43rd Annual International Symposium on Computer Architecture (ISCA), June 2016, pp. 367–379. 2, 10, 12, 25, 26, 45

[13] J. Luo, J. Wu, and W. Lin, "Thinet: A filter level pruning method for deep neural network compression," *CoRR*, vol. abs/1707.06342, 2017. [Online]. Available: http://arxiv.org/abs/1707.06342 2

[14] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," *CoRR*, vol. abs/1704.04861, 2017. [Online]. Available: http://arxiv.org/abs/1704.04861 2

[15] X. Zhang, X. Zhou, M. Lin, and J. Sun, "Shufflenet: An extremely efficient convolutional neural network for mobile devices," *CoRR*, vol. abs/1707.01083, 2017. [Online]. Available: http://arxiv.org/abs/1707.01083 2

[16] S. Anwar, K. Hwang, and W. Sung, "Fixed point optimization of deep convolutional neural networks for object recognition," *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 1131–1135, 2015. 3, 6

[17] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. S. Bernstein, A. C. Berg, and F. Li, "Imagenet large scale visual recognition challenge," *CoRR*, vol. abs/1409.0575, 2014. [Online]. Available: http://arxiv.org/abs/1409.0575 3

[18] Y. LeCun and C. Cortes, "MNIST handwritten digit database," 2010. [Online]. Available: http://yann.lecun.com/exdb/mnist/ 6

[19] F. Li and B. Liu, "Ternary weight networks," *CoRR*, vol. abs/1605.04711, 2016. [Online]. Available: http://arxiv.org/abs/1605.04711 6

[20] M. Courbariaux and Y. Bengio, "Binarynet: Training deep neural networks with weights and activations constrained to +1 or -1," *CoRR*, vol. abs/1602.02830, 2016. [Online]. Available: http://arxiv.org/abs/1602.02830 6

[21] A. Krizhevsky, V. Nair, and G. Hinton, "Cifar-10 (canadian institute for advanced research)." [Online]. Available: http://www.cs.toronto.edu/~kriz/cifar.html 6

[22] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," *CoRR*, vol. abs/1502.03167, 2015. [Online]. Available: http://arxiv.org/abs/1502.03167 19

[23] johnjohnlin, "Nicotb, a python-verilog co-simulation framework." [Online]. Available: https://github.com/johnjohnlin/nicotb 44