FPRaker: Exploiting Fine-Grain Sparsity to Accelerate Neural Network Training

by

Omar Alaaeldin Mohamed Amin Mohamed Awad

A thesis submitted in conformity with the requirements
for the degree of Masters of Applied Science
Graduate Department of Electrical and Computer Engineering
University of Toronto

Abstract

FPRaker: Exploiting Fine-Grain Sparsity to Accelerate Neural Network Training

Omar Alaaeldin Mohamed Amin Mohamed Awad
Masters of Applied Science
Graduate Department of Electrical and Computer Engineering
University of Toronto
2020

This thesis presents *FPRaker*, a processing element for composing training accelerators. Training manipulates floating-point data and multiply-accumulate (MAC) operations constitute the bulk of its computations. *FPRaker* boosts performance and energy-efficiency by skipping ineffectual computations during training. *FPRaker* processes the operands' significand of each MAC as a series of signed-powers-of-two, or *terms*. This exposes ineffectual work that can be skipped: encoded values have few terms and some can be discarded as they would fall outside the accumulator's precision. Over 9 studied networks, *FPRaker* is 1.5× faster and 1.4× more energy-efficient compared to a baseline accelerator with conventional TensorCore-like tiles under iso-compute-area constraints. We demonstrate that *FPRaker* delivers additional benefits when training incorporates pruning [82], quantization [19] and methods that use a different accumulator precision per layer [98]. Finally, we propose a memory compression technique for exponents of floating-point values that exploits the narrow value distribution during training using base-delta compression reducing off-chip memory bandwidth.

Acknowledgements

I would like to deeply thank my supervisor, Prof. Andreas Moshovos, for his patient guidance, feedback, and support throughout the two years of my Masters at the University of Toronto. I consider myself lucky to be one of his students and I'll be forever indebted to him for the so many things he taught me both on the technical and personal levels.

I would like to express my gratitude to my defense committee members, Prof. Gennady Pekhimenko and Prof. Paul Chow for their valuable feedback and suggestions for further improving this work. My grateful thanks to Mostafa Mahmoud with whom I constantly collaborated throughout my research. I also would like to thank my lab mates: Isak Edo, Ali Hadi Zadeh, Sayeh Sharify, Alberto Delmas, Milos Nikolic, Dylan Malone Stuart, Kevin Siu, Ciaran Bannon, Eugene Sha. Our technical discussions have made this journey more fruitful and enjoyable.

Finally, special thanks to my parents and sisters for always being there for me. Their continuous support and love were major sources of encouragement and drive throughout those two years. Thank you for always being by my side.

Contents

List of Figures

List of Tables

List of Abbreviations

ML	Machine Learning
DL	Deep Learning
DNN	Deep Neural Network
CNN	Convolution Neural Network
GPU	Graphics Processing Unit
TPU	Tensor Processing Unit
RL	Reinforcement Learning
ILSVRC	ImageNet Large Scale Visual Recognition Challenge
INQ	Incremental Network Quantization
LRN	Local Response Normalization
BN	Batch Normalization
LSTM	Long-Short Term Memory
FC	Fully-Connected
BNORM	Batch Normalization
NCF	Neural Collaborative Filtering
BERT	Bidirectional Encoder Representation from Transformers
SNLI	Stanford Natural Language Inference
FP	Floating-Point
MSB	Most Significant Bit
RNE	Rounding to nearest Even
MAC	Multiply-Accumulate
FMAC	Fused Multiply-Accumulate
MLP	Multi-Layer Perceptron
PACT	PArameterized Clipping acTivation function
PE	Processing Element
ReLU	Rectified Linear Unit
PReLU	Parameterized ReLU
ResNet	Residual Network
RNN	Recurrent Neural Network
SGD	Stochastic Gradient Descent
DRAM	Dynamic Random Access Memory

Chapter 1

Introduction

1.1 Motivation

Recent advances in Deep Learning (DL) have led to breakthroughs in a wide variety of applications, achieving unprecedented accuracy in tasks that were considered unfeasible for computing machines to perform. In a very short time, DL has become the state-of-the-art method for numerous applications such as image classification [64], speech recognition [43], natural language processing [107], object detection [35], recommendation systems [47], language modelling [27], neural machine translation [18], and scene understanding [74]. Interestingly, although the the basic foundations of DL and the core algorithm used to train deep neural networks (DNNs) were established back in the 1980s [96], it was only recently that we have seen a resurgence of interest in DL with the availability of large training data sets such as ImageNet [97] and computing hardware that delivers the necessary compute power to practical DNN models. These two factors have fueled further innovation and as a result more high accurate but at the same time more demanding DNN models continue to emerge.

The continuous need to achieve higher accuracy on different tasks has led to increasing both the depth (number of layers) and breadth (layer size) of the neural networks. Whereas a decade ago, the then state-of-the-art neural networks could be trained on a commodity server within a few hours, today training the best neural networks has become an exascale class problem whose compute and memory requirements demand weeks of compute time even with state-of-the-art training methods and on top-of-the-line hardware [9, 115]. Current state-of-the-art neural networks require many graphics processing units (GPUs) [1], or specialized accelerators such as the TPU [60], DaVinci [70], Cerebras CS1 [2], or Gaudi [3] for them to be trained within practical time limits, e.g., training ResNet50 on ImageNet in an hour requires 250 P100 GPUs [38]. Tuning neural networks for state-of-the-art execution time or accuracy, e.g., via hyperparameter tuning [108] or neural architectural search [31], further amplifies the cost of training. Training neural networks at edge devices is also required to refine or augment already existing models with user-specific information and input, known as adaptive learning [13, 57, 122]. The tradeoffs for training solutions differ depending on the type of targeted device segment. Specifically, operating and maintenance costs, latency, throughput, and node count are major considerations for data centers. At the edge, however, energy and latency are major considerations. Regardless of the application, increasing performance and energy efficiency of training would be of value.

It then comes as no surprise that efforts to increase the performance and energy efficiency of training have been considerable. Furthermore, such methods can often be used in combination. Such methods include the following: Distributed training partitions the training workload across several computing nodes taking advantage of data,

model, or pipeline parallelism [53, 85, 101, 106, 109]. Timing communication and computation can further reduce training time [45, 48, 59, 91, 120]. Dataflow optimizations to facilitate data blocking and to maximize data reuse reduce the cost of on- and off-chip accesses within the node maximizing reuse from lower cost components of the memory hierarchy [15, 34]. Another family of methods reduces the footprint of the intermediate data needed during training. For example, in the simplest form of training, all neuron values produced during the forward pass are kept to be used during backpropagation. Batching and keeping only one or a few samples instead reduces this cost [76]. Lossless and lossy compression methods further reduce the footprint of such data [8, 58, 73, 95, 100, 120]. Finally, selective backpropagation methods alter the backward pass by propagating loss only for some of the neurons [111] thus reducing work.

While the aforementioned method improve execution time and energy efficiency during training, the need to boost energy efficiency during inference is leading to techniques that further increase computation and memory needs of training. This includes works that perform network pruning and quantization during training. Pruning zero weights creates an opportunity for reducing work and model size during inference. Quantization produces models that use shorter datatypes that are more energy-efficient to compute such as 16b, 8b or 4b fixed-point values. Parameter Efficient Training [82], Memorized Sparse Back-propagation [125] are examples of recent pruning methods. PACT [19] and outlier-aware quantization [88] are state-of-the-art training time quantization methods. Network architecture search techniques also increase training time as they adjust the model's architecture [31].

As a result, training remains an exascale class problem and further improvements are needed. In general, the bulk of the computations and data transfers during training is for performing multiply-accumulate (MAC) operations during the forward and backward passes of training. I observe that during training, many ineffectual MAC operations occur naturally and for a variety of models. The goal of this thesis is to design a processing element (PE) that exploits ineffectual work to reduce energy consumption and to improve execution time.

Since inference is a subcomponent of training and since it also performs many MAC operations, in our quest to design a processing element (PE) that can eliminate ineffectual work during training, we may attempt to build upon the numerous proposals that exploit ineffectual work during *inference*. I highlight some of those approaches that are most relevant to this work. Any MAC operation where any of the two input operands is zero, be it the activation or the weight, is ineffectual. The first class of accelerators that exploit ineffectual operations rely on that zeros occur naturally in the activations of many models especially when they use ReLU [7, 16, 41, 62]. Another class of accelerators target zeros in weights. Some zero weights naturally occur in DNN models. However, their presence can be amplified through pruning, a method that converts many of the weights into zeros. Pruning typically requires training from scratch or extra training epochs [22, 42]. There are several accelerators that target pruned models e.g., [6, 17, 37, 40, 41, 65, 75, 87, 124, 128]. Another class of designs benefit from reduced value ranges whether these occur naturally or result from quantization. This includes bit-serial designs [29, 61, 67, 69, 103], and designs that support many different datatypes such as BitFusion [104]. Finally, another class of designs targets *bit-sparsity* where by decomposing multiplication into a series of shift-and-add operations they expose ineffectual work at the bit-level [6, 25, 102]. These designs exploit the distribution of values in DNNs that, at the bit-level, especially after Booth-Encoding [10] is skewed towards values that comprise only a few powers of two.

While we can draw from the experience gained from the aforementioned designs for inference, training presents us with different challenges. First, is the *datatype*. While models during inference work with fixed-point values of relatively limited range, the values training operates upon tend to be spread over a much larger range. Accordingly, training implementations use floating-point arithmetic with single-precision IEEE floating point arithmetic (FP32) being sufficient for virtually all models. Other datatypes that facilitate the use of more energy- and area-efficient multiply-accumulate units compared to FP32 have been successfully used in training many models with minimal

to no loss in the model accuracy. These include the brain floating-point format (bfloat16), and 8b or smaller floating-point formats [23,24,39,49,63,117,118]. Moreover, since floating-point arithmetic is a lot more expensive than integer arithmetic, mixed datatype training methods use floating-point arithmetic only sparingly [23,28,80,86]. Despite these proposals, FP32 remains today the standard fall-back format, especially for training on large and challenging datasets. Many of the aforementioned inference accelerators rely upon phenomena that emerge due to the use of fixed-point arithmetic during inference: as a result of its limited range and the lack of an exponent, the fixed-point representation used during inference gives rise to zero values (too small a value to be represented), zero bit prefixes (small value that can be represented), and bit sparsity (most values tend to be small and few are large). In contrast, FP32 can represent much smaller values, its mantissa is normalized, and whether bit sparsity exists has not been demonstrated.

Second, is the *computation structure*. Inference operates on two tensors, the weights and the activations, performing per layer a matrix/matrix or matrix/vector multiplication or pairwise vector operations to produce the activations for the next layer in a feed-forward fashion. Training includes this computation as its *forward* pass which is followed by the *backward* pass that involves a third tensor, the gradients (see Section 2.1.4 for a primer on training). Most importantly, the backward pass uses the activation and weight tensors in a different way than the forward pass (the set of input values that contribute to an output is different between the two passes), making it difficult to pack them efficiently in memory, more so to remove zeros as done by inference accelerators that target sparsity. Related to computation structure, third, is *value mutability* and *value content*. Whereas in inference the weights are static, they are not so during training. Furthermore, training initializes the network with random values which it then slowly adjusts. Accordingly, one cannot necessarily expect the values processed during training to exhibit similar behavior such as sparsity or bit-sparsity. More so, for the gradients which are values that do not appear at all during inference.

In this thesis, we target the processing elements for the MAC operations and propose designs that exploit *ineffectual* work that occurs naturally during training and whose frequency is amplified by quantization, pruning, and selective backpropagation. Specifically, we observe that if we decompose each multiplication as a series of shift-and-add operations of signed powers of two, only a few powers appear per input operand (we target the activations) and some of those powers can be discarded if we take into account the precision of the floating-point format used and the current value of the accumulator or other products in a MAC unit that performs multiple MAC operations in parallel.

1.2 Contributions

This thesis makes the following contributions:

- Demonstrates that a large fraction of the work performed in training is ineffectual. To expose this ineffectual work we decompose each multiplication into a series of single-bit multiply-accumulate operations. This reveals two sources of ineffectual work: First, more than 85% of the computations are ineffectual since one of the inputs is zero. Second, the combination of the high dynamic range (exponent) and the limited precision (mantissa) often yields values which are non-zero, yet too small to affect the accumulated result, even when using extended precision (e.g., trying to accumulate 2^{-64} into 2^{64}).
- Proposes *FPRaker*, a processing tile for *training* accelerators which exploits both bit-sparsity and out-of-bounds computations to boost performance and energy efficiency. *FPRaker* comprises several adder-tree based processing elements — each processing element performs multiple MAC operations all accumulated

into a single output — organized in a grid so that it can exploit data reuse both spatially and temporally. The processing elements multiply multiple value pairs concurrently and accumulate their products into an output accumulator. They process one of the input operands per multiplication as a series of signed powers of two hitherto referred to as *terms*. The conversion of that operand into powers of two is performed on the fly; all operands are stored in floating point form in memory.

FPRaker has the following characteristics:

- It does not affect numerical accuracy. The results it produces throughout the training process adhere to results produced by conventional floating-point arithmetic, and hence achieves similar final validation accuracy for the trained model.
- It skips ineffectual operations that would result from zero mantissa bits and from out-of-range intermediate values.
- Despite individual MAC operations taking more than one cycle, *FPRaker*'s computational throughput is higher compared to conventional floating-point units; given that *FPRaker* processing elements are much smaller we can fit more of them in the same area.
- It naturally supports shorter mantissa lengths thus rewarding innovation in training methods with mixed or shorter datatypes [98, 117]. It does so while not requiring that the methods be universally applicable to all models.
- *FPRaker* allows us to choose which tensor input we wish to process serially per layer. This allows us to target those tensors that have more sparsity depending on the layer and the pass (forward or backward).

• Presents a simple, low-overhead memory encoding for floating-point values that relies on the value distribution that is typical of deep learning training. I observed that consecutive values across channels have similar values and thus exponents. Accordingly, I encode the exponents as deltas for groups of such values using the base-delta compression scheme [90]. I use this encoding when storing and reading values off chip, thus further reducing the cost of memory transfers.

1.3 Thesis Organization

The rest of the thesis is organized as follows. Chapter 2 provides an introduction to DL, discusses the core operations in DNN models, and presents the DNN models and datasets studied in this thesis. It also discusses how inference and training tasks for DNNs differ in terms of datatypes, dataflow, amount of computations, and memory organization. Chapter 3 overviews related work for deep neural network acceleration on both the software and hardware levels. Further, it overviews floating-point arithmetic and previous work on bit-serial floating-point multiply-accumulate hardware. Chapter 4 presents the proposed *FPRaker* training accelerator and evaluates its execution time performance and energy efficiency. Chapter 5 summarizes the contributions of this thesis and presents directions for future work.

Chapter 2

Background

This chapter provides the essential background needed to understand the work presented. Section 2.1 first discusses the fundamentals of machine learning (ML) and how it relates to deep learning (DL). It then discusses the limitations of some conventional machine learning methods such as linear and logistic regression models which motivate the introduction of neural networks. Next, it discusses the main computation layers in DNNs. We then end this section by discussing the two main tasks in DL which are inference and training, highlighting the differences between the two tasks. Section 2.2 presents the networks studied in this thesis and discusses the application of each network and the dataset used for training it. Section 2.3 offers a summary for this chapter.

2.1 Introduction to Deep Learning

Computers have been traditionally programmed using conventional algorithms where the programmer defines and codes the algorithm to solve a specific problem. This is sufficient for problems for which we can define an analytical solution. For example, a palindrome can be described as a sequence that reads the same backwards as forward. Therefore, we can write a simple program using a conventional programming language to classify any given sequence whether it is a palindrome or not. Now let us think of a different example, such as whether an image contains a dog or a cat, it becomes very challenging to define an analytical solution. This is also the case for any problem that deals with unstructured data such as image classification, object detection, natural language processing, and speech recognition. **Machine Learning** (ML) is a set of algorithms and techniques to solve such problems, where the algorithm learns how to solve the problem by learning and extracting features from the input raw data.

The most popular methods in machine learning can be classified into three categories: supervised, unsupervised, and reinforcement learning. **Supervised Learning** is the most popular paradigm of ML [68], where the ML model learns an input-to-output mapping function given enough input training examples with their correct labeled outputs. Initially, the model predictions will be completely random. The goal of the training process is to minimize the cost function which is usually the error between the model's predicted output value and the labeled true output value over the training examples, and hence increase the model's accuracy of predicting new inputs that were never presented the model. Supervised learning can be classified into two types: *classification*, and *regression* models. Classification models try to map an input to a given set of output classes. Models with two, or more output classes are known as *binary*, and *multiclass classification* models, respectively. Regression models targets problems with continuous output values. Examples of supervised learning are linear and logistic regression [5].

Unsupervised learning features no output labels during training. This makes it especially important since most data in the world is unlabeled. Instead, the model learns the probability distribution of the input data. This however makes it hard to define a clear error function to measure the accuracy of the model. Examples of unsupervised learning are clustering algorithms like k-means [71] and Expectation-Minimization (EM) clustering with Gaussian Mixture Models (GMM) [77].

Reinforcement learning (RL) is a learning approach based on interaction [112], where the models learns by receiving either a positive or negative reinforcement after taking each action. The model needs to discover actions that maximize the reward. The most popular application for RL is teaching machines how to play games, e.g. AlphaGo beating the world champion in the board game Go in 2016. Recently, RL has been applied in chip design to automate the placement and routing process, e.g., generating optimized placements for Google's TPU that outperforms manually-designed counterparts in less than 6 hours [4].

2.1.1 From Linear Regression to Neural Networks

The simplest supervised learning algorithm is linear regression. It takes an n-dimensional input vector $\mathbf{x} \in \mathbb{R}^n$ and outputs a scalar value $\hat{y} \in \mathbb{R}$ as the prediction for the labeled output value $y \in \mathbb{R}$. The model parameters are defined as a weight vector $\mathbf{w} \in \mathbb{R}^n$ as shown in Equation 2.1.

$$\hat{y} = \mathbf{w}^\intercal \mathbf{x} \tag{2.1}$$

During the training process, the model optimizes its parameters to minimize the cost function $J(\mathbf{w})$ commonly defined as the mean squared error between the prediction $\hat{y}$ and the labeled output y over m training examples. Equation 2.2 shows $J(\mathbf{w})$ defined as the squared L^2-norm.

$$J(\mathbf{w}) = \frac{1}{m} \sum_{i=1}^{\infty} (\hat{y}_i - y_i)^2 \tag{2.2}$$

Equation 2.3 shows the vectorized form of Equation 2.2 with output predictions vector $\hat{\mathbf{y}} \in \mathbb{R}^m$, output labeled vector $\mathbf{y} \in \mathbb{R}^m$, and a matrix of n-dimensional input vector $\mathbf{X} \in \mathbb{R}^{m \times n}$.

$$J(\mathbf{w}) = \frac{1}{m} \|\hat{\mathbf{y}} - \mathbf{y}\|_2^2 = \frac{1}{m} \|\mathbf{X}\mathbf{w} - \mathbf{y}\|_2^2 \tag{2.3}$$

To find the values of $\mathbf{w}$ that minimize the cost function J($\mathbf{w}$), an analytical solution can be derived for small number of training examples by solving the gradient of J with respect to $\mathbf{w}$ ($\frac{\partial J}{\partial \mathbf{w}}$) as follows:

$$\because \nabla_{\mathbf{w}} J(\mathbf{w}) = 0 \tag{2.4}$$

$$\therefore \frac{1}{m} \nabla_{\mathbf{w}} \|\mathbf{X}\mathbf{w} - \mathbf{y}\|_2^2 = 0 \tag{2.5}$$

$$\therefore \frac{1}{m} \nabla_{\mathbf{w}} (\mathbf{X}\mathbf{w} - \mathbf{y})^\intercal (\mathbf{X}\mathbf{w} - \mathbf{y}) = 0 \tag{2.6}$$

$$\therefore \nabla_{\mathbf{w}} (\mathbf{w}^\intercal \mathbf{X}^\intercal \mathbf{X}\mathbf{w} - 2\mathbf{w}^\intercal \mathbf{X}^\intercal \mathbf{y} + \mathbf{y}\mathbf{y}^\intercal) = 0 \tag{2.7}$$

$$\therefore (2\mathbf{X}^\intercal \mathbf{X}\mathbf{w} - 2\mathbf{X}^\intercal \mathbf{y}) = 0 \tag{2.8}$$

$$\therefore \mathbf{w} = (\mathbf{X}^\intercal \mathbf{X})^{-1} \mathbf{X}^\intercal \mathbf{y} \tag{2.9}$$

However such closed-form solution does not scale well with the increasing the size of the input feature vector, since the computational complexity of inverting the $(\mathbf{X}^\intercal \mathbf{X})$ matrix is $O(n^2 m)$. Such optimization problems are typically solved using the iterative algorithm of **gradient descent**, where the model parameters $\mathbf{w}$ are optimized to minimize the cost function J in incremental steps scaled with the learning rate α as shown in Equation 2.10. A very small α would require a large number of steps before the model converges to a global minimum, while a very large α can lead the model to oscillate, or even overshoot the global minimum and diverge. Parameters like the learning rate are called **hyperparameters** which act as tuning knobs for the learning algorithm during training, and they are typically set manually prior to training.

$$w_i = w_i - \alpha \frac{\partial J(\mathbf{w})}{\partial w_i} \tag{2.10}$$

A visualization of the gradient descent algorithm is shown in Figure 2.1.

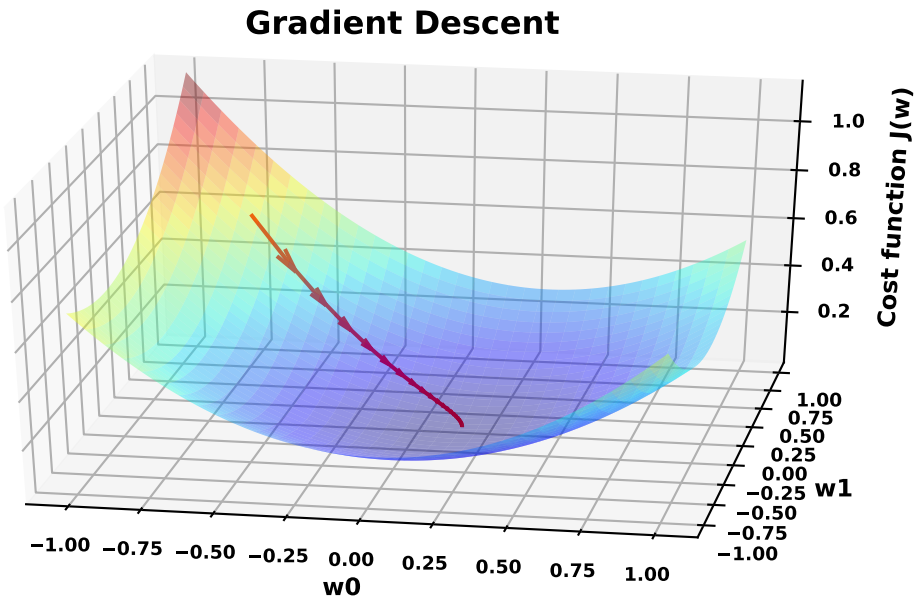

Figure 2.1: Gradient descent algorithm starts with randomly initialized weights $(w0, w1)$ and incrementally updates the weights in the direction of the steepest slope in the cost function till it converges to the global minimum.

In binary classification problems, **logistic regression** is typically used where the output of the model is a discrete value $y \in \{0, 1\}$ representing the probability of the input belonging to class-1, unlike linear regression which has a continuous output. As shown in Equation 2.11, logistic regression can be seen as a linear regression followed by the sigmoid non-linear activation function σ.

$$\hat{y} = P(y = 1 | \mathbf{x}; \mathbf{w}) = \sigma(\mathbf{x}^\intercal \mathbf{w}), \quad \sigma(z) = \frac{1}{1 + e^{-z}} \tag{2.11}$$

There is no closed-form analytical solution for logistic regression due to the non-linearity of the sigmoid function, and hence gradient descent is used to optimize the model parameters. Logistic regression uses the negative log-likelihood instead of the mean squared error as the cost function as shown in Equation 2.12, since the latter

would result in a non-convex cost function which can lead the model to be trapped in a saddle point and never converge to the global minimum.

$$J(\mathbf{w}) = -\frac{1}{m}[\mathbf{y}^\intercal \log(\mathbf{X}\mathbf{w}) + (\mathbf{1}-\mathbf{y})^\intercal \log(\mathbf{1}-\mathbf{X}\mathbf{w})] \tag{2.12}$$

2.1.2 Neural Networks

Neural networks are a class of machine learning techniques that are loosely inspired by the biological neural networks in the human's brain, and currently achieve the state-of-the-art performance on a variety of applications including image classification, object detection, natural language processing, speech recognition, recommendation systems, and language modeling. Figure 2.2 shows a simple example of a 2-layer neural network, where the output value of each node, called an activation, in the output layer is a weighted sum of all the input activations, hence known as fully-connected (FC) layer, followed by a Rectified Linear Unit (ReLU) activation function. The connections between activations are called weights. Equation 2.13 shows the computations performed per output activation, where N and M are the number of input and output activations, respectively, f is a non-linear activation function such as the ReLU function, and B_j is a bias term added per output activation O_j.

$$\forall j \in \{0,1,...,M-1\} \ : \ O_j = f(\sum_{i=0}^{N-1} A_i \times W_{i,j} + B_j) \tag{2.13}$$

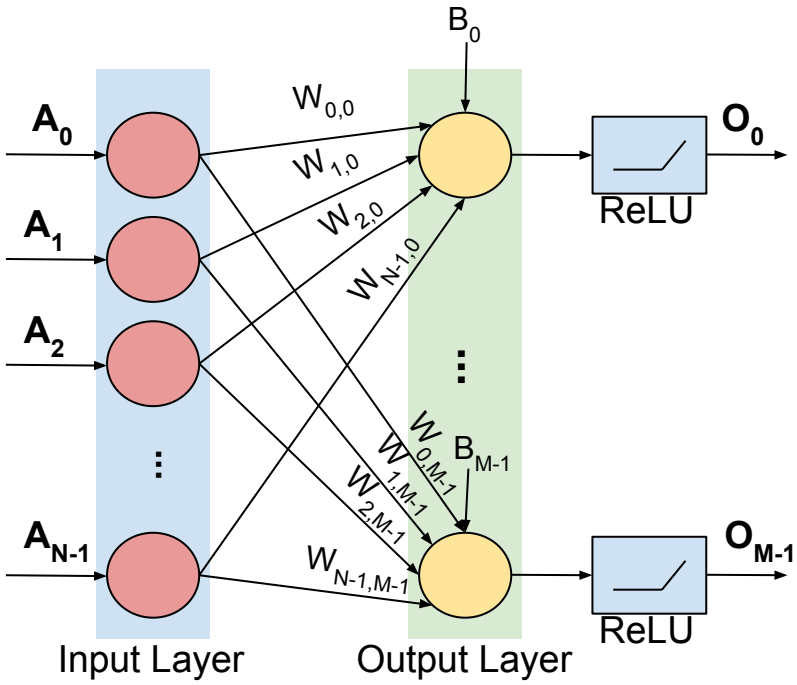

Figure 2.2: A simple 2-layer neural network with ReLU activation function

Deep neural networks are organized as layers arranged in a pipelined fashion where the output values of one layer are fed as inputs to the next layer as shown in Figure 2.3. For clarity, weight connections between layers are represented as a single connection labeled $\mathbf{W}^i$, where i is the layer number. Increasing the number of hidden layers,

i.e., a deeper neural network, can increase the model capacity to perform more complex tasks [93], however this can have diminishing returns to the model accuracy due to overfitting [114], i.e., model learns training set too well (high training accuracy) and does not perform well on new input samples (low validation accuracy).

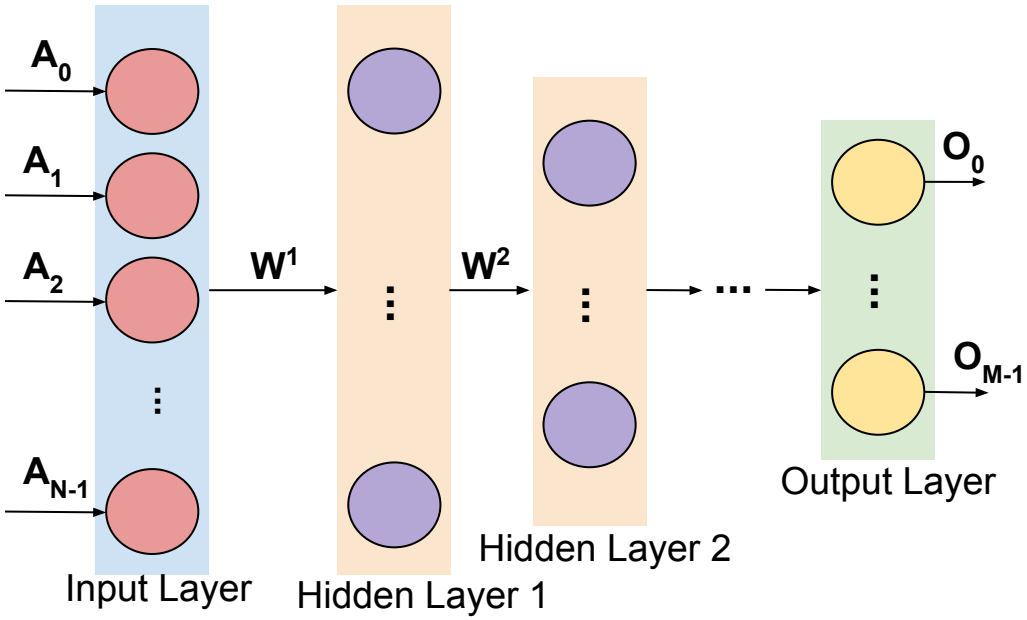

Figure 2.3: Deep neural network with several hidden layers

Multi-layer neural networks with fully-connected activations between consecutive layers are known as **Multi-layer Perceptrons** (MLP). While MLP networks can be a good fit for some applications, they usually have expensive computation and memory requirements due to the large number of model's weight parameters. **Convolution neural networks** (CNNs) address this problem by introducing convolution layers. In fully-connected layers, each output activation is calculated through all activations in an input image. In some applications, the input features are limited in spatial scope. Hence, convolution layers are used as feature detectors to capture correlations among inputs activations that gives rise to the detection of a feature. This allows convolution layers to reduce the models weight parameters by sharing the same weight filters across multiple output activations, where it looks for the desired feature by inspecting only a limited set of input activations at a time in a strided window fashion. Accordingly, using convolution layers leads to a reduced computation and memory requirements compared to fully-connected layers with the same number of input and output activations. CNNs are typically used in applications such as image classification, object detection, image segmentation and captioning achieving the state-of-the-art accuracy for these applications.

Standalone CNNs, however, are not a good fit for applications that require keeping track with previous inputs such as natural language processing, neural machine translation, recommendation systems, and speech recognition. **Recurrent neural networks** (RNNs) address this issue. RNNs can capture temporal information and dependencies in sequential input data streams using their internal memories. One issue with conventional RNNs is learning long-term dependencies in the input data due to the vanishing gradient problem [52]. **Long-Short Term Memory** (LSTM) networks were introduced to address this issue by adding a gated cell which enables them to capture long-term dependencies by controlling the information flow and keeping track with only important data.

This thesis focuses on accelerating both convolution and fully-connected layers. CNNs are dominated by convolution layers, while fully-connected layers usually account for less than 10% of the overall execution time.

However, fully-connected layers dominate transformer-based, recommendation, and LSTM-based networks. The four main types of layers in DNN are convolution, fully-connected, normalization, and pooling layers. Since fully-connected layers were discussed previously, the following discussion reviews convolution, normalization, and pooling layers.

2.1.2.1 Convolution Layer

Figure 2.4 shows a convolution layer where input and output activations are organized as 2D tensors, each representing a feature, stacked along the *channel* dimension to construct 3D tensors. The 3D tensors have the shape of (H,W,C) representing the height, width, and number of channels of the tensor, respectively. Convolution layers have K weight filters, denoted as $f^0, f^1, ..., f^{K-1}$. Each filter is applied in a sliding window fashion across the horizontal and vertical dimensions of the 3D input tensor with some stride S. The application of one filter produces a 2D output tensor (1 channel). The 2D output tensors, each produced by a different filter, are stacked along the channel dimension to form a 3D output tensor that is fed as an input to the next layer. Assuming an input activation tensor with dimensions $C \times H \times W$, K 3D filters with dimensions $C \times H_f \times W_f$, sliding window of stride S, the resulting output activation tensor is of size K, H_o, W_o where the output height is calculated as:

$$H_o = \frac{H - H_f}{S} + 1 \tag{2.14}$$

and the output width is calculated as:

$$W_o = \frac{W - W_f}{S} + 1 \tag{2.15}$$

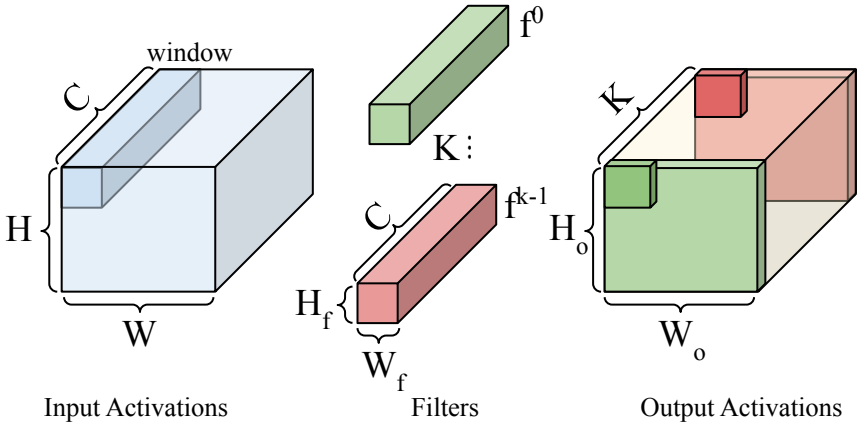

Figure 2.4: 3D convolution layer

Each output activation is resulting from the inner product of a filter and a sub-array of the input activation tensor with similar size, known as window. If a, and f^n are, respectively, the input activation, n^{th} filter, then the corresponding output activation is computed as:

$$o(n,y,x) = \sum_{c=0}^{C-1} \sum_{j=0}^{H_f-1} \sum_{i=0}^{W_f-1} f^n(c,j,i) \times a(c, j+y \times S, i+x \times S) \tag{2.16}$$

After computing the inner product in Equation 2.16, a bias term is added to the output activation which then

passes by a non-linear activation function.

2.1.2.2 Depthwise Separable Convolution Layer

The convolution layer captures two types of correlation in the input activation tensor: intra-channel, and inter-channel correlation. While regular convolution layers captures these two types of correlation in one step, the depthwise separable convolution extract such correlations using two convolution layers instead: 1) a layer with *C* **depthwise separable filters**, i.e. each input activation channel is convolved with a 2D filter to extract the intra-channel correlation as shown in Figure 2.5, and 2) a layer with *N* **pointwise separable filters** to extract the inter-channel correlation as shown in Figure 2.6. Stacking these two types of convolution layers combines the intra-channel and inter-channel correlations. The advantage of the depthwise separable convolution is the fewer computations compared to the corresponding regular convolution which make it a good fit for mobile and embedded applications. However, networks with depthwise separable convolution are usually less accurate compared to using regular convolutions.

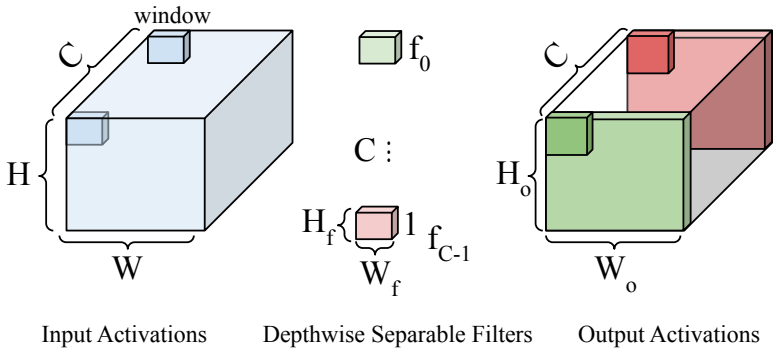

Figure 2.5: Depthwise Separable Convolution

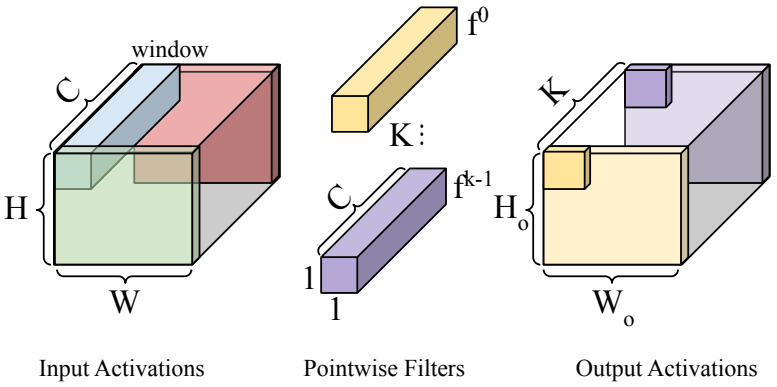

Figure 2.6: Pointwise Separable Convolution

Figure 2.7: Depthwise Separable Convolution followed by Pointwise Separable Convolution

2.1.2.3 Normalization Layers

Local Response Normalization (LRN) is a compute heavy layer that was used in early CNN models such as AlexNet [64] to normalize each element in the input activation tensor with respect to the other elements at the same

location in the adjacent N input activation tensors using the formula in Equation 2.17.

$$o(c,y,x) = a(c,y,x) \times (1 + \alpha \sum_{i=c-\frac{N}{2}}^{c+\frac{N}{2}} (a(i,y,x))^2)^{-\beta} \tag{2.17}$$

The function of LRN layer is create *lateral inhibition* of the output activation values especially when using an unbounded activation function such as ReLU [51]. This layer, however, is removed in newer models such as the ResNets [46] and sometimes replaced by a **batch normalization** (BNORM) layer followed by scaling, which reduced the required training steps while achieving similar accuracy. Equation 2.18 shows the computation for the BNORM layer where μ and σ^2 are statistically computed over the training dataset, while γ and β are learned during training [56].

$$o(c,y,x) = \gamma \times (\frac{a(c,y,x) - \mu}{\sqrt{\sigma^2}}) + \beta \tag{2.18}$$

After training a network, the $\mu, \sigma^2, \gamma, \beta$ are all constants in the BNORM layer. Accordingly, Equation 2.18 can be simplified to the following Equation where $A = \frac{\mu}{\sqrt{\sigma^2}}$ and $B = \beta - \frac{\gamma\mu}{\sqrt{\sigma^2}}$:

$$o(c,y,x) = A \times a(c,y,x) + B \tag{2.19}$$

2.1.2.4 Pooling Layer

A Pooling (POOL) layer acts as a down-sampling function such that the input activation tensor of size $N_x \times N_y$ is reduced in size with the same number of channels, input and output activation tensors. Examples of POOL layers are the Max-POOL and Average-POOL layers where each element in the output activation tensor represent the maximum and average value of a window of size $K_p \times K_p$ in the input activation tensor respectively.

2.1.2.5 Activation Layer

Table 2.1 shows the most commonly used activation functions in DNNs. ReLU is the most commonly used activation function and especially in CNNs, since it requires low computation and leads to a faster training process.

Table 2.1: Common Activation Functions

Function	**Definition**
ReLU	ReLU(x) = $\begin{cases} x, if x > 0 \\ 0, otherwise \end{cases}$
Sigmoid	$\sigma(x) = \frac{1}{1+e^{-x}}$
Hyperbolic Tangent	$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$
Parameterized ReLU (PReLU)	PReLU(x) = $\begin{cases} x, if x > 0 \\ ax, otherwise \end{cases}$ [for a = 0.01 $\implies$ LeakyReLU(x)]

2.1.3 Inference

Inference applies the acquired knowledge of a trained neural network model given a new input to infer the result. The computations involved in inference were previously discussed in Section 2.1.2. As shown previously, inference involves two input tensors, the weights and the activations, which are used in only one computation, typically a matrix-matrix multiplication or a matrix-vector multiplication. Thus, the two tensors can be laid out in memory in a

way that serves a specific access pattern facilitating data parallel, and thus energy-efficient, fetching and execution. Inference is performed using fixed-point datatypes such as 16b, 8b, or of even shorter lengths in some cases such as 4b or 1b. Inference is usually performed on a single input at a time. This is done to minimize response latency in time critical applications or user interactive ones. In some applications, several inputs can be processed together, a process referred to as batching. This results in higher reuse of weights and can reduce overall memory traffic. However, this comes at the expense of individual input latency.

This thesis focuses on training whose forward propagation pass is identical to inference. The next section discusses this in more detail.

2.1.4 Training

The goal of training is to adjust the weight values so that the network can perform the desired task with high accuracy. There are several variants of training and without loss of generality we describe the basic process. Training starts by setting the weights to some "random" values. Then training repeatedly goes over a large set of annotated samples adjusting the weight values per batch. These samples are inputs for which the desired output is already known. The samples are processed in batches. For each batch training performs the forward pass and calculates a set of outputs. These outputs are "compared" with the annotated, desired outputs. The comparison takes the form of a *loss* function which is a measure of how "far" the current outputs are from the desired ones. The loss values are the propagated backwards through the network where they are used to adjust the weights. The back propagation happens per sample, whereas the changes to the weights are aggregated over the whole batch and applied once. We describe the forward and backward passes for convolutional layers first.

The **forward pass** is the same as inference. A sample is processed though a series (really a directed acyclic graph) of layers to produce the final output. Each layer accepts as input a 3D array I (feature maps containing activations) and a set K of 3D arrays W (filters containing weights) and produces an output array O (output activations). O becomes the input for the next in series layer and the process repeats. The width and height of I is usually much larger than that of the W. The convolutional layer applies 3D convolution of I with the W filters to produce the output. The convolution is applied in a sliding window fashion. Each convolution of one W with one window of I produces a single O value.

The **backward pass** updates the weights and the activations. Informally, compared to what the output activation of each layer was during the immediately preceding forward pass, we now have an updated activation which is expressed as a small change in the output value (gradient). The goal is to propagate this small change to the inputs to "nudge" them so that the error might be reduced during the next forward pass. Accordingly, to calculate the change for each input we have to backpropagate the changes from *all* outputs it influenced. In a way we have to perform the "mirror" computation that was performed for the forward pass. Each layer convolves its output gradients with the weights to produce the *input gradients* to be fed to the preceding layer. The layer also convolves its output gradients with its input activations to calculate the *weight gradients*. The per layer weight gradients are accumulated across the training samples within a mini-batch and used to update the weights once per mini-batch, or iteration, as described by Equation Eq. (2.20), where i is the layer number, t is the iteration number, α is the learning rate, and S is the mini-batch size.

$$W_i^{t+1} = W_i^t - \alpha * \sum_{s=0}^{S} Gw_i^s / S \tag{2.20}$$

Table 2.2 describes the operations in more detail. For clarity, Figs 2.8-2.10 show the operations only for the convolutional layers. A fully-connected layer can be treated as a special-case convolutional layer where all input tensors are of equal size.

Table 2.2: Training Process: Processing of one training sample. Weights are updated per batch (see text). The notation used for activations, weights, activation gradients, weight gradients is respectively $A^{S/L}_{c,x,y}, W^{L,F}_{c,x,y}, G^{S/L}_{c,x,y}, Gw^{S/L,F}_{c,x,y}$, where S refers to the training sample, L refers to the network layer, F is the weight filter, c is the channel number, and x, y are the 2D spatial coordinates. The stride is denoted as st.

Forward Pass

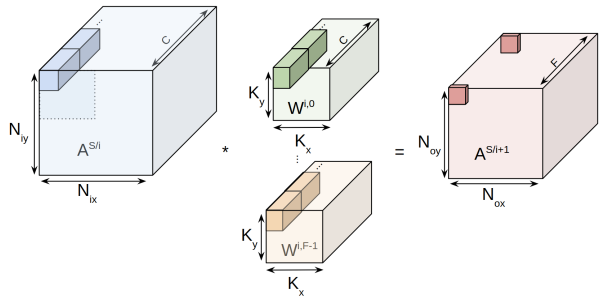

Figure 2.8: Forward convolution

Convolutional Layer: A sliding-window 3D convolution is performed between the input activations and each of the weight filters to produce one channel in the output activations:

$$A^{S/i+1}_{oc,ox,oy} = \sum_{ci=0}^{C} \sum_{xi=0}^{Kx} \sum_{yi=0}^{Ky} A^{S/i}_{ci,ox*st+xi,oy*st+yi} * W^{i,oc}_{ci,xi,yi} \quad (2.21)$$

Fully-Connected: Each filter produces one output activation:

$$A^{S/i+1}_{oc} = \sum_{ci=0}^{C} A^{S/i}_{ci} * W^{i,oc}_{ci} \quad (2.22)$$

Backward Pass

Input Gradients

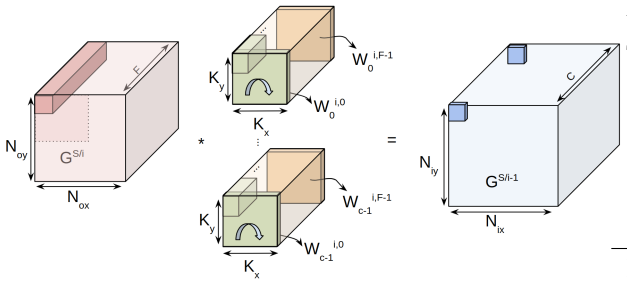

Figure 2.9: Calculating input gradients

Convolutional Layer: A sliding-window 3D convolution is performed between a *reshaped* version of the filters with the activation gradients from the subsequent layer. The filters are reconstructed channel-wise and rotated by 180 degrees and the activation gradients are dilated by the stride st.

$$G^{S/i-1}_{oc,ox,oy} = \sum_{ci=0}^{F} \sum_{xi=0}^{Kx} \sum_{yi=0}^{Ky} G^{S/i}_{ci,ox+xi,oy+yi} * W_{rotated}{}^{i,ci}_{oc,xi,yi} \quad (2.23)$$

Fully-Connected: The filters are reconstructed as above. No dilation of the activation gradients.

$$G^{S/i-1}_{oc} = \sum_{ci=0}^{F} G^{S/i}_{ci} * W^{i,ci}_{oc} \quad (2.24)$$

Weight Gradients

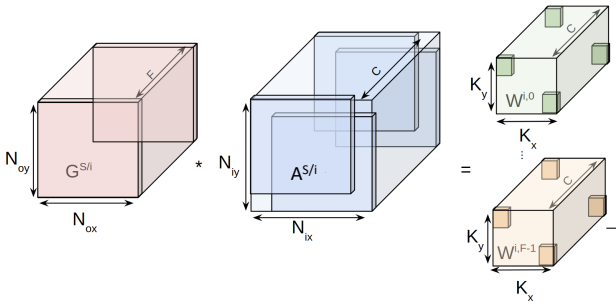

Figure 2.10: Calculating weight gradients

Convolutional Layer: The weight gradients are accumulated across batch samples. Per sample, it is calculated as a 2D convolution between the input activation and the output gradients which are dilated according to the stride.

$$Gw^{total/i,f}_{oc,ox,oy} = \sum_{si=0}^{S} \sum_{xi=0}^{Nox} \sum_{yi=0}^{Noy} G^{si/i}_{f,xi,yi} * A^{si/i}_{oc,ox+xi,oy+yi} \quad (2.25)$$

Fully-Connected: Each weight gradient is a scalar product of the input activation and the gradient of the output activation it affects accumulated over samples.

$$Gw^{total/i,f}_{oc} = \sum_{si=0}^{S} G^{si/i}_{f} * A^{si/i}_{oc} \quad (2.26)$$

There are three major differences between the training and inference phases. First, is the **datatype**. While models during inference work with fixed-point values of relatively limited range, the values training operates upon tend to be spread over a large range. Accordingly, training implementations use floating-point arithmetic with single-precision IEEE floating point arithmetic (FP32) being sufficient for virtually all models. Other datatypes that facilitate the use of more energy- and area-efficient multiply-accumulate units compared to FP32 have been successfully used in training many models. These include bfloat16, and 8b or smaller floating-point formats [23, 24, 39, 49, 63, 117, 118]. Moreover, since floating-point arithmetic is a lot more expensive than integer arithmetic, mixed datatype training methods use floating-point arithmetic only sparingly [23, 28, 80, 86]. Despite these proposals, FP32 remains today the standard fall-back format, especially for training on large and challenging datasets.

Second, is the **computation structure**. Inference operates on two tensors, the weights and the activations, performing per layer a matrix/matrix or matrix/vector multiplication or pairwise vector operations to produce the activations for the next layer in a feed-forward fashion. Training includes this computation as its *forward* pass which is followed by the *backward* pass that involves a third tensor, the gradients. Most importantly, the backward pass uses the activation and weight tensors in a different way than the forward pass, making it difficult to pack them efficiently in memory.

Related to computation structure, third, is **value mutability** and **value content**. Whereas in inference the weights are static, they are not so during training. Furthermore, training initializes the network with random values which it then slowly adjusts. Accordingly, one cannot necessarily expect the values processed during training to exhibit similar behavior such as sparsity or bit-sparsity. More so, for the gradients which are values that do not appear at all during inference.

Fourth, is the **batch size**. Inference is typically performed with a single input example at a time, i.e. batch size of one. Training computes the gradients on a *mini-batch* of examples before updating the weights. A mini-batch is typically 64-256 examples. Mini-batches also increase training performance by computing each layer for a batch of examples at a time, which increases weight reuse for each training pass. The concept of propagating multiple examples at once is known as *batching*.

Table 2.3: Models Studied

Model	Application	Dataset
SqueezeNet 1.1	Image Classification	ImageNet [97]
VGG16	Image Classification	ImageNet [97]
ResNet18-Q	Image Classification	ImageNet [97]
ResNet50-S2	Image Classification	ImageNet [97]
SNLI	Natural Language Inference	SNLI Corpus [11]
Image2Text	Image-to-Text Conversion	im2latex-100k [119]
Detectron2	Object Detection	COCO [72]
NCF	Recommendation	ml-20m [44]
Bert	Language Modeling	WMT17 [30]

2.2 Networks Studied

In this thesis, we study the performance and energy-efficiency of the proposed accelerator on networks across a wide variety of applications. Table 2.3 lists the models studied. Our workloads includes models that have been optimized

using two widely used methods: quantization and pruning. Quantization changes the datatype used by the model to one that is more energy efficient to store and compute with. For example, rather than using 32b floating-point numbers, after quantization the model uses fixed-point numbers that are also narrower, e.g., 4b. Pruning converts some weights to zero which opens up opportunities for model compression (avoids storing the zero weights) and computation reduction (omit the MAC operations where the weight is zero). ResNet18-Q is a variant of ResNet18 trained using *PACT* [19] which quantizes both activations and weights down to 4b during training. ResNet50-S2 is a variant of ResNet50 trained using *dynamic sparse reparameterization* [83] which targets sparse learning that maintain high weight sparsity throughout the training process while achieving accuracy levels comparable to baseline training. SNLI performs natural language inference and comprises of fully-connected, LSTM-encoder, ReLU, and dropout layers. Image2Text is an encoder-decoder model for image-to-markup generation. We study three models of different tasks from the **MLPerf training benchmark** [78]: 1) Detectron2: an object detection model based on Mask R-CNN, 2) NCF: a model for collaborative filtering, and 3) Bert: a transformer-based model using attention.

For our measurements we sample one randomly selected batch per epoch over as many epochs as necessary to train the network to its originally reported accuracy (up to 90 epochs were enough for all).

We next provide additional information about the networks used as benchmarks in this thesis and including the corresponding training datasets we used.

2.2.1 Image Classification

The task of recognizing an object within an image is called *image classification*, which is the domain that started the resurgence of attention to deep learning. The networks studied for this task are convolutional neural networks (CNNs). The dataset used to train image classification networks in this thesis is *ImageNet* [97], which is a large dataset of colored images initially developed for the ImageNet Large Scale Visual Recognition Challenge (ILSVRC). The ILSVRC 2012 dataset used in this thesis consists of 1.2 million training images, 50,000 validation images, and 1,000 classes. The dataset has two well-known performance metrics to report accuracy: *top-1* and *top-5*. The *top-1* metric represents the percentage of example for which the predicted class with the highest probability matches the target label. The *top-5* is a less strict metric representing the percentage of examples whose any of its five predicted classes with highest probability matches the target label.

2.2.1.1 SqueezeNet 1.1

SqueezeNet [54] is a CNN that has 50$\times$ less parameters than AlexNet with an accuracy comparable to AlexNet on the ImageNet dataset. A compressed Squeezenet has a small memory requirement of less than 0.5MB, i.e. 510$\times$ smaller than AlexNet without compression. The building block of SqueezeNet is the *fire module*, which contains two layers: a squeeze layer and an expand layer. The squeeze layers are made up of 1×1 filters (as a form of model compression) capturing correlation along the channel dimension, while the expand layers consist of a mix of 1×1 and 3×3 filters to capture the spatial correlation. A SqueezeNet stacks a number of fire modules and a few pooling layers. The squeeze layer and expand layer keep the same feature map size, while the former reduces the depth to a smaller number (by having a small number of filters), and the latter increases it (by having a larger number of filters). The squeeze layer, also known as the bottleneck layer, and the expansion behavior are common in neural architectures.

2.2.1.2 VGG16

The Visual Geometry Group (VGG) [107] at the University of Oxford proposed several image classification networks that achieve high accuracy on the ImageNet dataset. Among the proposed networks is VGG16, which achieved the second best top-5 accuracy in the ILSVRC 2014 competition with 92.7% accuracy. VGG16 consists of 13 3×3 convolution layers with unit stride followed by 3 fully-connected layers.

2.2.1.3 ResNet

One of the major challenges in training DNNs is the *vanishing gradient* problem. This happens when backpropagating an error gradient through many layers with certain activation functions such as tanh and sigmoid, where the values of gradients shrink towards zero. This behavior effectively prevents weight updates, and hence makes the network harder to train.

The Residual Network (ResNet) [46] was developed to solve the vanishing gradient problem. It uses residual blocks each of which contains a bypass layer which allows the gradients to reach deeper layers more easily. In residual blocks, the output of a layer is connected as a direct input to the next and to the second next layer through a bypass connection. ResNet50 is a variant of ResNet, which has 49 convolution layers followed by a single fully-connected layer. ResNet was the winning network architecture in ILSVRC 2015 with 96.4% top-5 accuracy. ResNet18 is a smaller variant of ResNet that has 17 convolution layers followed by a fully-connected layer.

2.2.2 Object Detection

Object detection is the task of identifying objects in images or videos. Typically, the task of an object detection algorithm is to predict a bounding box and a class for each object in the given input image. We study *Detectron2* [121] which is a recent object detection network from the MLPerf training benchmark, which is based on the *Mask-RCNN* model. The *Base-RCNN with feature pyramid network* (Base-RCNN-FPN), also known as *Faster-RCNN-FPN*, is the basic bounding box detector used in the *Mask-RCNN* model. *Base-RCNN-FPN* is the de-facto standard detector that can detect multi-scale objects, i.e., from tiny to large objects, with high accuracy. We trained Detectron2 using *COCO* [72] dataset, which is a large-scale object detection, segmentation, and captioning dataset.

2.2.3 Scene Understanding

Scene understanding, also known as captioning, is the task of generating a human-level natural language description given an input image. We study *Image2Text* [116], network which deals with image captioning as a multimodal translation task using a sequence-to-sequence recurrent neural networks (RNN) model for image caption generation. Instead of representing the whole image using a CNN model, the *Image2Text* network represents the input image as a sequence of detected objects to serve as the source sequence of the RNN model. We train the *Image2Text* using the *im2latex-100k* [119] dataset, which is a pre-built dataset for OpenAI's task for image-2-latex system.

2.2.4 Recommendation Systems

We study the Neural Collaborative Filtering (NCF) [47] network that tackles *collaborative filtering*, which is the key problem in recommendation, on the basis of implicit feedback. The past interactions of users and items are recorded in a matrix format called "user-item interaction matrix". Previously, the user interaction with the item features resorted to matrix factorization and applied an inner product on the latent features of users and items. NCF replaces the inner product with a multi-layer perceptron (MLP) that learns the user-item interaction function, which

is a function representing the user-item interaction matrix. We trained NCF using the *ml-20m* [44] dataset, which contains 20 million ratings and 465,000 tag applications applied to 27,000 movies by 138,000 users.

2.2.5 Natural Language Modeling

The goal of a language model is to calculate the probability of a token, i.e., a sentence or a sequence of words. Language model acts as the grammar of a language as it gives the probability of the word that will follow. We studied two networks for this task: Bert-Base, and SNLI.

2.2.5.1 Bert

Bidirectional Encoder Representations from Transformers (BERT) [27] is a trained Transformer Encoder stack with twelve encoders in the Base version, and twenty-four encoders in the Large version compared to only 6 encoder layers in the original Transformer model. BERT encoders have larger feed-forward networks with 768 and 1024 nodes in Base and Large models respectively, and more attention heads with 12 and 16 heads in the *Base* and *Large* models respectively. In this thesis, we study the BERT-Base model. BERT was trained on the Wikipedia and Book Corpus [79], a dataset containing +10,000 books of different genres. BERT achieves high accuracy on a wide variety of language tasks such as question answering [94], named entity recognition [113], and sentiment analysis [84]. Each task requires minimal fine-tuning starting from a pre-trained model, i.e., only certain hyperparameters need to be tuned while most hyperparameters stay the same.

2.2.5.2 SNLI

The aim of the Stanford Natural Language Inference (SNLI) [11] model is to determine if a premise sentence is entailed, neutral, or contradicts a hypothesis sentence, e.g., “A basketball game with multiple people playing” entails “Some people are playing a sport”, while it contradicts “A man is playing a sport alone”. The SNLI model is a stack of three 200-dimensional FC layers each followed by a tanh activation function with the first layer taking the concatenated sentence representations as input and the final layer feeding a softmax classifier, all trained jointly with the sentence embedding model itself using the SNLI training corpus.

2.3 Summary

This chapter provides an introduction to machine learning (ML) with a focus on deep learning (DL), and discussed the limitation of linear regression models which motivated the development of neural networks. It reviewed the different layers that are used by DNNs, namely fully-connected, convolution, depthwise separable convolution, normalization, pooling, and activation layers. Moreover, this chapter discussed the two main tasks in deep learning: inference and training, and explained the differences between the two tasks in terms of the used datatypes, dataflow, and batch size thus explaining that the tradeoffs and requirements for training accelerators are different than those for inference-only ones. Finally, it discussed the studied networks in this thesis and their corresponding training datasets.

Chapter 3

Related Work

This chapter discusses the related work to this thesis. Section 3.1 overviews popular software techniques to accelerate DNN training. Section 3.2 discusses previous work on bit-serial inference accelerators. Section 3.3 provides an overview on floating-point arithmetic and previous work on bit-serial floating-point multiply-accumulate units. Section 3.4 reviews previous work on accelerators targeting removing the floating-point ineffectual computations and compares it to *FPRaker*. Section 3.5 provides a summary for this section.

3.1 Software Approaches to Accelerate Training

This section overviews software approaches to reduce the computation and memory requirements of DNN training. The following subsections discuss some of these approaches.

3.1.1 Pruning

The goal of pruning is to convert some, ideally unimportant or ineffectual, weight values to zero. Dynamic sparse reparameterization [82], sparse momentum [26], and eager pruning [123] are recent training-time pruning methods that achieve high sparsity levels with minimal or no effects on output accuracy. Dynamic sparse reparameterization is a technique based on dynamically allocating non-zero parameters during training through network structural exploration allowing direct training of sparse models without having to pre-train a large dense model while achieving comparable accuracy. Sparse momentum is an algorithm which uses exponentially smoothed gradients (momentum) to identify essential layers and weights for reducing the training error efficiently.

3.1.2 Quantization

The goal of quantization is to reduce the data width that will be used during inference and/or training. During training, quantization effectively clips what would otherwise be values of low magnitude into zeros. A low-precision training algorithm was proposed by Coubariaux et al. [21] to reduce the computation overhead in DNNs which is applicable for fixed-point, floating-point, and dynamic fixed-point operations. The proposed training algorithm performs both training forward and backward propagations with low precision, while weight updates are performed using high precision. DoReFa-Net [127] is a training method for CNNs using low precision for activations, weights and gradients. Using 2b activations, 1b weights and 6b gradients, DoReFa-Net achieved a top-1 accuracy for AlexNet comparable to training using FP32 format. Value-aware quantization [89] is a technique

applied for both training and inference which represents most of the data using aggressively reduced precision while large data values are separately handled using higher precision. PArameterized Clipping acTivation function (PACT) [19] is a technique that enables training DNNs using ultra low precision, i.e., 2-4b, with no or minimal loss of accuracy. PACT uses an activation clipping parameter that is optimized during training in order to find the optimal quantization scale for activations.

3.1.3 Selective Back-propagation

Selective back-propagation is an approach to reduce the computational requirement in back-propagation during training. MeProp [111] sparsifies the back-propagation by selecting only a small subset of the full gradients to update the model parameters. MeProp back-propagates only the top-k gradient values in terms of magnitude. DropBack [36] is another technique that constraints the total number of weight updates during back-propagation to those with the highest total gradients. Weights that are not updated are discarded and their initial values are regenerated at every access to avoid storing them in memory. This technique reduces the memory requirement and number of off-chip memory accesses for training.

3.2 Bit-Serial Hardware Accelerators for Inference

Bit-skipping processing of multiply-accumulate operations has been proposed before for inference. Bit-Pragmatic is a data-parallel processing element that performs such bit-skipping of one operand side [6] whereas Laconic does so for both sides [102]. Since these methods target inference only they work with fixed-point values. Since we found that there is little bit-sparsity in the weights during training, we focused on the Bit-Pragmatic's approach. Converting a fixed-point design to floating-point is a non-trivial task to start with. Regardless, converting Bit-Pragmatic into floating-point resulted in an area-expensive unit that performs poorly under iso-compute-area constraints (see Section 4.7.1.1). Specifically, compared to an *optimized* Bfloat16 bit-parallel processing element (see Section 4.2) that performs 8 MAC operations, under ISO-compute constraints, an optimized accelerator configuration using the Bfloat16 Bit-Pragmatic PEs is on average 1.72× slower and 1.96× less energy efficient. In the worst case, the Bfloat16 bit-pragmatic PE was 2.86× slower and 3.2× less energy efficient. The Bfloat16 Bit-Pragmatic PE is 2.5× smaller than the bit-parallel PE, and while we can use more such PEs for the same area, we cannot fit enough of them to boost performance via parallelism as required by all bit-serial and bit-skipping designs.

At the microarchitectural level, *FPRaker* is a floating-point PE where the processing of values differs significantly from that of fixed-point values. The similarity between *FPRaker* and Bit-Pragmatic is limited in the use of the 2-stage shifting technique, which we adapted to reduce the area of our adder tree. Laconic uses a unique processing element which is designed for fixed-point arithmetic only. Extending it to support floating-point operation is left for future work.

Compared to inference, training performs more computation and requires a larger volume of data to be kept around. While in inference, activations and weights are used only once during training, they are used twice and more importantly the dataflow is different between these two uses. This challenges several optimizations that are otherwise possible in inference. For example, inference accelerators targeting sparsity such as Cambricon-X [124] or SCNN [87] prepack values in memory according to a pre-determined dataflow to eliminate zero values. Unfortunately, this is not directly compatible with training as the order in which values will be accessed during backpropagation needs to be different than that during the forward pass (inference).

3.3 Floating-Point Arithmetic

3.3.1 Fused Multiply-Accumulate

Fused multiply-accumulate (FMAC) units combine the multiplication and accumulation steps by skipping the normalization step after multiplication and directly adding the resulting product with the accumulator. FMAC reduces the area and latency of the multiply-accumulate operation. Figure 3.1 shows a block diagram of the steps in the FMAC operation.

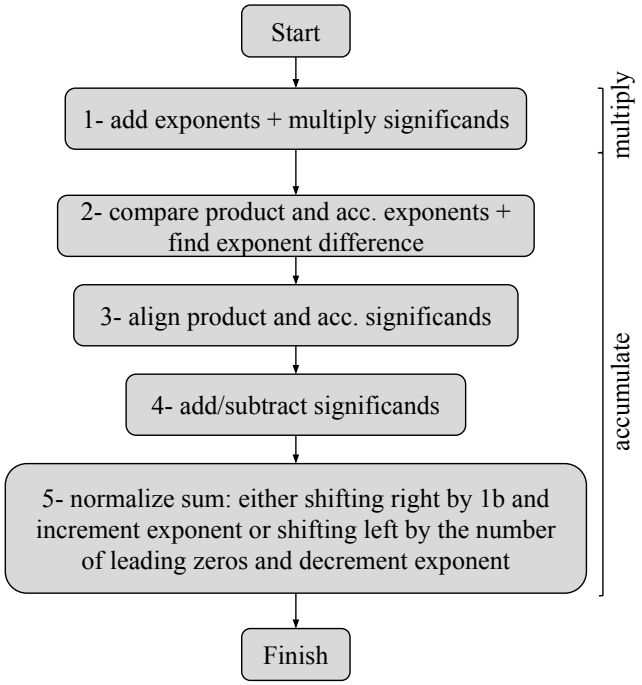

Figure 3.1: Block diagram of floating-point fused multiply-accumulate (FMAC)

3.3.2 Bit-Serial Multiply-Accumulate Units

Bit-serial arithmetic circuits have long been used in embedded applications such as in signal processing with the emphasis being on reducing cost and where performance was acceptable. The closest related designs are the single multiplier of Shinde and Salankar [105] and the single MAC of Chau et al. [14] which targets fault tolerance. Both use a multiplier stage while *FPRaker* is a SIMD MAC that processes Booth terms across multiple value pairs eliminating the need for a multiplier stage and bares little similarity at the microarchitectural level with the aforementioned designs.

3.4 Accelerators Targeting Floating-Point Ineffectual Computations

Feinberg et al. [32] proposes a memristive accelerator that supports double-precision floating-point *in-situ* matrix-vector multiplication for scientific computing, e.g., krylov subspace methods. *FPRaker*, however, is a purely digital DNN training accelerator supporting Bfloat16 precision and implemented using standard CMOS technology. Performing floating-point addition on a memristor crossbar requires converting the floating-point values

into aligned fixed-point values increasing sparsity by extra padding bits. The memristive accelerator removes ineffectual computation on the crossbar-granularity by having different crossbar sizes per cluster and mapping blocks of effectual computations in the multiplicand matrix to similar-size crossbars through a pre-processing step. Inefficiently-mapped blocks are instead handled by the local processor. *FPRaker*, however, performs regular floating-point addition and does not require fixed-point conversion. *FPRaker* removes more ineffectual computations by skipping zero and out-of-bound terms per value through its term-serial processing.

3.5 Summary

This chapter first presented an overview on the software techniques to accelerate DNN training. Then, it discussed related work on inference accelerators based on fixed-point bit-serial MAC units. Further, it presented an overview on the floating-point fused multiply-accumulate operation and previous work on floating-point bit-serial MAC units. Finally, it reviewed previous work on floating-point accelerators that targets removing the ineffectual computations and and compares it to *FPRaker*.

Chapter 4

FPRaker: DNN Training Accelerator

This chapter presents *FPRaker*, a hardware accelerator for training DNNs that exploits bit-sparsity to boost the training performance and energy-efficiency. Section 4.1 shows the inherent bit-sparsity in traces collected during training which motivates *FPRaker* and shows its potential performance for each computation phase during training. Section 4.2 presents the baseline accelerator used in this thesis. Section 4.3 discusses the approach of *FPRaker* to exploit bit-sparsity by breaking down and serializing the multiplication operation. Section 4.4 presents the design of *FPRaker* accelerator and related architectural optimizations to reduce the compute area and increase performance. Section 4.5 discusses the spatial correlation between FP values during training, and proposes a new compression method for encoding the exponents in FP values based on the base-delta compression technique to save the off-chip memory transfers. Section 4.6 discusses the data organization and supply to the processing elements of both the *FPRaker* and baseline accelerators to keep them busy. Section 4.7 discusses the evaluation of *FPRaker* against the baseline accelerator. Finally, section 4.8 summarizes the work in this thesis and gives concluding remarks.

4.1 Ineffectual Work During Training

This section motivates *FPRaker* by measuring the work reduction that is ideally possible with two related approaches: 1) by removing all MACs where at least one of the operands are zero (value sparsity, or simply sparsity), and 2) by decomposing multiplications into a series of shift-and-add operations effectively processing only the non-zero bits of the mantissa (bit-sparsity).

The bulk of work during training is due to three major operations per layer:

$$Z = I \cdot W \qquad (4.1) \qquad \frac{\partial E}{\partial I} = W^T \cdot \frac{\partial E}{\partial Z} \qquad (4.2) \qquad \frac{\partial E}{\partial W} = I \cdot \frac{\partial E}{\partial Z} \qquad (4.3)$$

For convolutional layers Eq. 4.1 describes the convolution of activations (I) and weights (W) that produces the output activations (Z) during forward propagation. There the output Z passes through an activation function before used as input for the next layer. Eq. 4.2 and 4.3 describe the calculation of the activation ($\frac{\partial E}{\partial I}$) and weight ($\frac{\partial E}{\partial W}$) gradients respectively in the backward propagation. Only the activation gradients are back-propagated across layers. The weight gradients update the layer's weights once per batch. For fully-connected layers the equations describe several matrix-vector operations. For other layers they describe vector or matrix-vector operations. For clarity, in the rest of this work we will refer to the gradients as G.

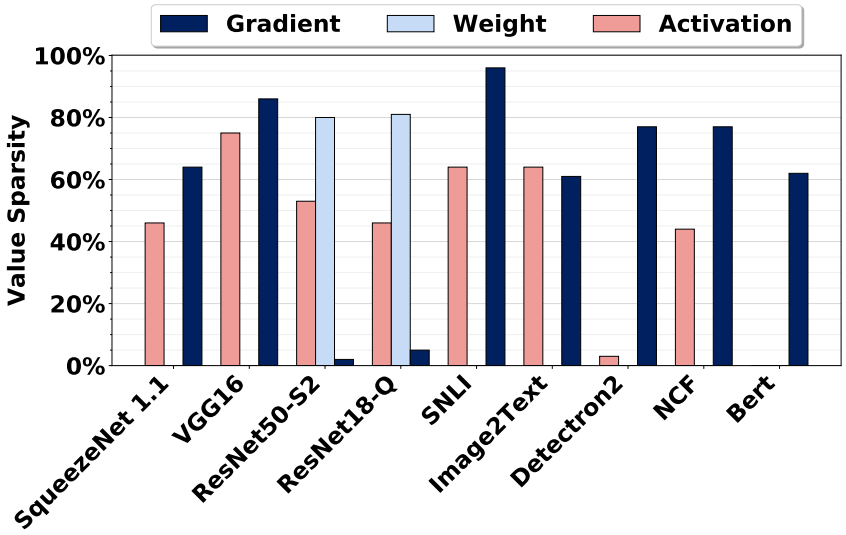

Figure 4.1: Value Sparsity in Tensors During Training.

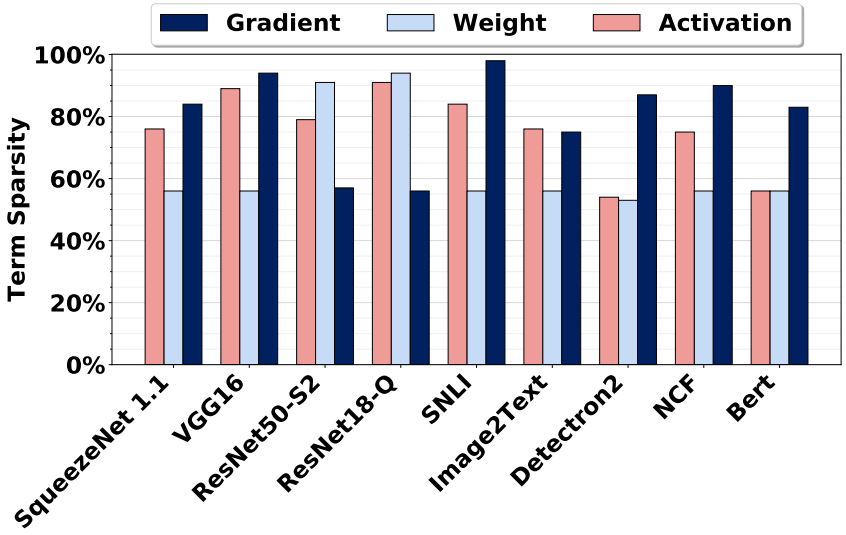

Figure 4.2: Term Sparsity in Tensors During Training.

Figure 4.1, and 4.2 show the value, and term-sparsity respectively for each of the three tensors (W, I, and G). Each value is weighted according to frequency of use. A value can be either a weight or an activation all of which are represented as floating-point values. Sparsity in this case refers to the fraction of operations where either the weight or the activation is zero. Such operations can be safely eliminated without affecting the final outcome. We use the term *term-sparsity* to signify that for these the measurements the mantissa is first encoded into signed powers of two using Canonical digit encoding which is a variation of Booth-encoding. For example, the activation A=(1.0111110) is encoded using just three signed powers of two or *terms*, $(+2^{-0}, +2^{-1}, -2^{-7})$. Bit-sparsity refers to the fraction of bits that are zero. For our example value, bit sparsity is $2/8$ as there are two bits that are zero of the total 8 mantissa bits. If we were to convert the multiplication with this values into a series of shift-and-add operations we would have to perform 6 of them instead of 8 if we were to process the bits directly. Encoding the mantissa using signed power of two (terms) results into 3 terms yielding a term sparsity of $5/8$. If we

were to decompose the multiplication into a series of shift-and-add operations with signed powers of two, we would need to perform just 3 of them. For simplicity, from this point on we will use term- and bit-sparsity interchangeably while referring to term-sparsity as defined above.

The activations in the image classification networks exhibit sparsity exceeding 35% in all cases. This is expected since these networks use the ReLU activation function which clips negative values to zero. However, sparsity in the weights is typically low and only some of the classification models exhibit sparsity in their gradients. For the remaining models, however, such as those for natural language processing, value sparsity is very low for all three tensors. Regardless, since some models do exhibit sparsity it may be worthwhile to investigate whether it is possible to exploit it during training. As explained in the introduction, this is a non-trivial task, as training is different than inference and exhibits dynamic sparsity patterns on all tensors and different computation structures during the backward pass.

Figure 4.2 shows that all three tensors exhibit high term-sparsity for all models regardless of the target application. Given that term-sparsity is more prevalent than value sparsity, and exists in all models in the rest of this work we investigate whether it is practically possible to exploit it during training. One such option is the *FPRaker* processing element which we present next.

Figure 4.3 reports the ideal potential speedup due to reduction in the multiplication work through skipping the zero terms in the serial input. We calculate the potential speedup over the baseline as:

$$Potential\ speedup = \frac{\#MAC\ operations}{term\ sparsity \times \#MAC\ operations} \tag{4.4}$$

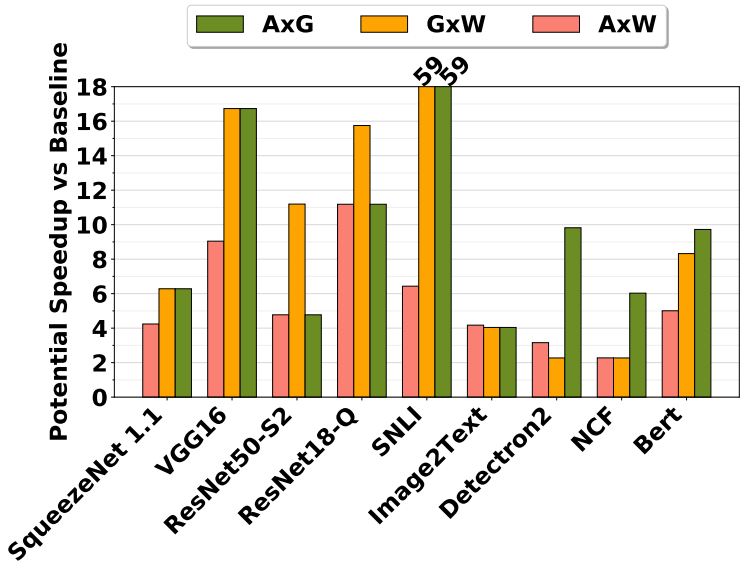

Figure 4.3: Performance improvement potential of exploiting term sparsity for the three training phases.

4.2 Bit-Parallel Baseline Accelerator

This section describes the baseline accelerator used in this thesis. We designed an efficient bit-parallel fused multiply-accumulate (MAC) unit with conventional multipliers as the baseline processing element (PE), which is shown in Figure 4.4. In a bit-parallel implementation all bits of a value are processed concurrently. The input and

output operands of the PE are in Bfloat16. Training DNNs with Bfloat16 format is known to converge to the same final accuracy as training with single precision (FP32) format and is widely adopted in industry, e.g., Google's TPU [60]. Each cycle, the PE multiplies 8 activation-weight pairs in parallel, reduces the resulting products using an adder-tree to a partial sum, and then accumulates the partial sum into an accumulation register that stores the partial output activation. The adder-tree amortizes the cost of *reading-modifying-writing* the partial output activation for each of the input activation-weight pairs. Hence, the adder-tree-based PE is more energy-efficient compared to a scalar MAC PE that performs a separate *read-modify-write* operation on the partial output activation for each input pairs. We chose 8 MACs per PE as this proves to be a good compromise between data parallelism and the need for extra computations that are needed when the relevant layer dimensions are not divisible by PE MAC width (layer fragmentation). A designer may chose to revisit this design parameter.

The PE can be decomposed into three stages: 1) *Multiply*: this stage has 8 bit-parallel floating-point multipliers. The constituent multipliers are both area and latency efficient, and are taken from the DesignWare IP library developed by Synopsys. Each multiplier outputs the product exponent (P_e) and significand (P_m). 2) *Align*: this stage aligns the product significands (P_m) with the current significand value stored in the accumulation register (ACC) according to their corresponding exponent values (P_e). To do so, a comparator-tree (MAX block) determines the maximum (e_{max}) among the products' exponents (P_e) and the current accumulator exponent (e_{ACC}). Figure 4.5 shows the MAX block. Each exponent product is then subtracted from the maximum exponent (e_{max}) to get the amount of shift (δ_e) for its corresponding significand (P_m). The MAX block outputs the "acc_shift" signal to align the accumulator register in case one of the products' exponents is larger than the current accumulator exponent. 3) *Accumulate*: this stage accumulates the result from the adder-tree with the partial output activation stored in the accumulation register. After each accumulation, the accumulation register value gets normalized and rounded using the rounding-to-nearest-even (RNE) rounding scheme.

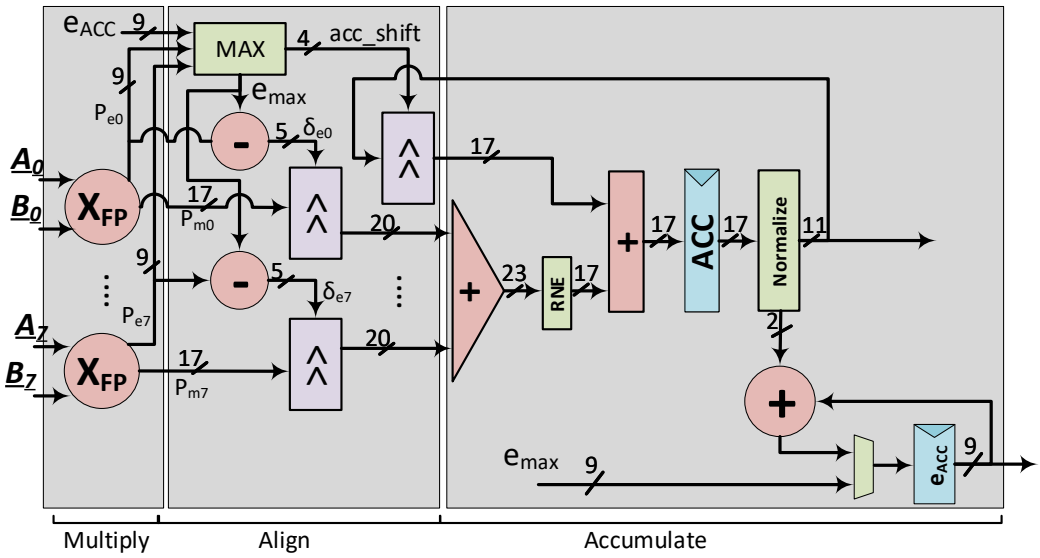

Figure 4.4: Baseline Processing Element

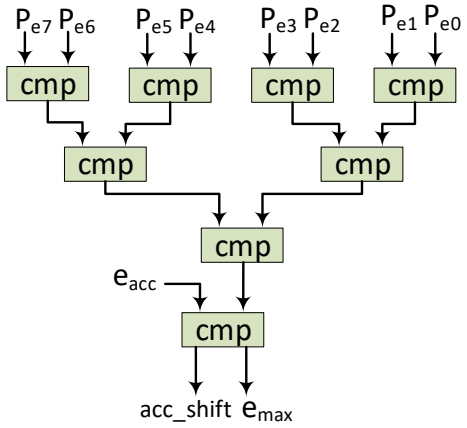

Figure 4.5: MAX block: a comparator-tree

To exploit data reuse spatially, the baseline is configured to have scaled-up GPU Tensor-Core-like tiles as shown in Figure 4.6, where 64 PEs are organized in a 8×8 grid and each PE performs 8 MAC operations in parallel. Smaller grid sizes, e.g. 4×4, can reduce the synchronization overhead among PEs of the same column, however, on the expense of reducing the spatial data reuse, and vice versa. The 8×8 grid was chosen so that a tile can perform full 8×8 vector-matrix multiplication.

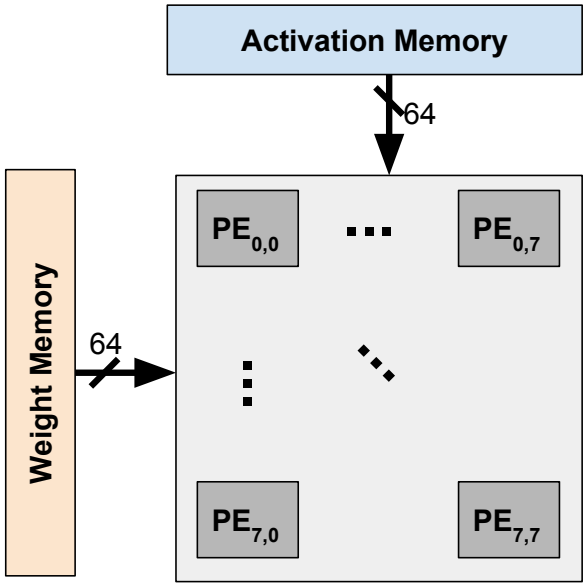

Figure 4.6: Baseline tile configuration

4.3 Exposing Ineffectual Work

FPRaker's goal is to take advantage of bit-sparsity in one of the operands used in the three operations performed during training (equations 4.1 through 4.3) all of which are composed of many MAC operations. We first explain how decomposing MAC operations into a series of shift-and-add operations can expose ineffectual work, providing us with the opportunity to save energy and time.

To expose ineffectual work during MAC operations we can decompose the operation into a series of "shift and add" operations, Let us first look at the multiplication. Let $A = 2^{A_e} \times A_m$ and $B = 2^{B_e} \times B_m$ be two values in

floating point, both represented as an exponent (A_e and B_e) and a significand (A_m and B_m) which is normalized and includes the implied "1.". Conventional floating point units perform this multiplication in a single step (sign bits are XORed):

$$A \times B = 2^{A_e+B_e} \times (A_m \times B_m) = (A_m \times B_m) \ll (A_e + B_e) \tag{4.5}$$

By decomposing A_m into a series p of signed powers of two A_m^p where $A = \sum_p A_m^p$ and $A_m^p = \pm 2^i$, we can instead perform the multiplication as follows:

$$A \times B = (\sum_p A_m^p \times B_m) \ll (A_e + B_e) = \sum_p B_m \ll (i + A_e + B_e) \tag{4.6}$$

For example, if A_m=1.0000001b, A_e=10b, B_m=1.1010011b and B_e=11b then we can perform $A \times B$ as two shift-and-add operations of $B_m \ll (10b+11b-0)$ and $B_m \ll (10b+11b-111b)$. A conventional multiplier would process all bits of A_m despite performing ineffectual work for the six bits that are zero.

However, this decomposition exposes further ineffectual work that conventional units perform as a result of the high dynamic range of values that floating point seeks to represent. Informally, some of the work done during the multiplication will result in values that will be out-of-bounds given the accumulator value. To understand why this is the case let us now consider not only the multiplication but also the accumulation. Let's assume that the product $A \times B$ will be accumulated into a running sum S and S_e is much larger than $A_e + B_e$. It will not be possible to represent the sum $S + A \times B$ given the limited precision of the mantissa. In other cases, some of the "shift-and-add" operations would be guaranteed to fall outside the mantissa even when we consider the increased mantissa length used to perform rounding, i.e., partial swamping. A conventional pipelined MAC unit can at best power-gate the multiplier and accumulator after comparing the exponents and only when the *whole* multiplication result falls out of range. However, it cannot use this opportunity to reduce cycle count. By decomposing the multiplication into several simpler operations, we can terminate the operation in a single cycle given that we process the bits from the most to the least significant, and thus boost performance by initiating another MAC earlier. The same is true when processing multiple $A \times B$ products in parallel in an adder-tree processing element. A conventional adder-tree based MAC unit can potentially power-gate the multiplier and the adder tree branches corresponding to products that will be out-of-bounds. The cycle will still be consumed. As we explain in the next section, a shift-and-add based unit will be able to terminate such products in a single cycle and advance others in their place.

4.4 Architecture

Section 4.4.1 shows the implementation of the *FPRaker* PE. Section 4.4.2 discusses a simplified execution example of the *FPRaker* processing element (PE). Section 4.4.3 explains how *FPRaker* time-multiplexes a single exponent block among multiple PEs, a key optimization for area- and energy-efficiency. Section 4.4.4 explains how multiple processing elements can be organized into a larger tile.

4.4.1 FPRaker Processing Element

In this section, we describe the implementation of the *FPRaker* PE. First, we discuss an initial baseline implementation of the *FPRaker* PE. Afterwards, we discuss limiting the accumulator bit-width through chunk-based accumulation to further reduce the area of the PE. Then, we discuss an optimization which takes advantage of the narrow value distribution during training to reduce the area of the shifting and adder-tree stages. This optimization limits the size of the shifters by decomposing shifting into two stages with a common shifter. Finally, this section

proposes a simple modification to the *FPRaker* PE to allow it to skip *out-of-bounds* bits during accumulation to further boost performance and energy-efficiency.

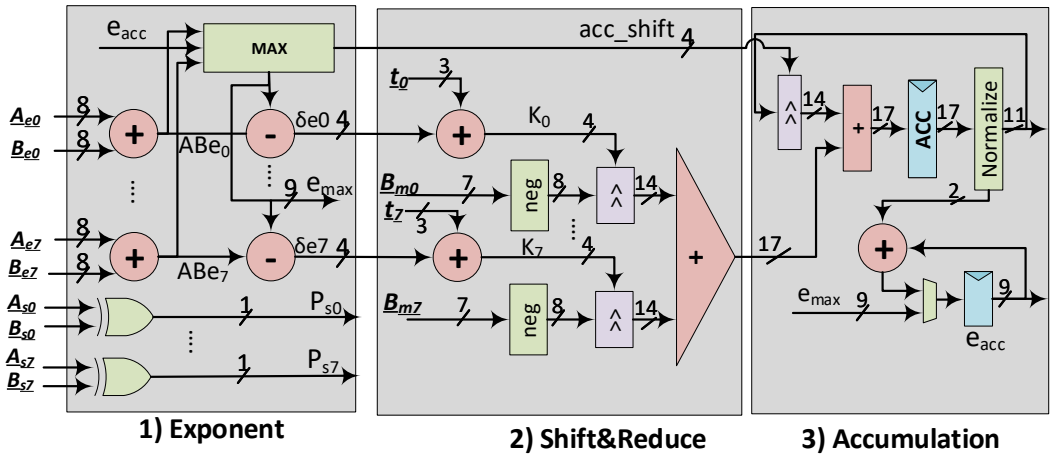

Figure 4.7: *FPRaker* PE: Baseline Design.

4.4.1.1 Baseline Design

The *FPRaker* PE performs the multiplication of 8 Bfloat16 (A,B) value pairs, concurrently accumulating the result into an output accumulator. The Bfloat16 format consists of a sign bit, followed by a biased 8b exponent, and a normalized 7b significand (mantissa). Figure 4.7 shows a baseline of the *FPRaker* PE design which performs the computation in 3 blocks: **exponent, reduction, and accumulation**. We describe an implementation where the 3 blocks are performed in a single cycle. We will build upon this design and modify it to construct a more area efficient tile comprising several of these PEs. Recall that the significands of each of the A operands are converted on-the-fly into a series of *terms* (signed powers of two) using canonical encoding. This encoding occurs just before the input to the PE. All values stay in bfloat16 while in memory. The PE will process the A values term-serially. The accumulator has an extended 13b significand where 1b is used for the leading 1 (hidden), 9b are used for extended precision following the **chunk-based accumulation** scheme as suggested by Sakr et al., [99] with a chunk-size of 64 which guarantees a training conversion accuracy within 1% of the final model accuracy trained with FP32 format on ImageNet dataset, plus 3b for rounding to nearest even. It has 3 additional integer bits following the hidden bit so that it can fit the worst case carry out from accumulating 8 products. In total, the accumulator has 16b, 4 integer, and 12 fractional.

The PE accepts 8 8-bit A exponents $A_{e0},...,A_{e7}$, their corresponding 8 3-bit significand *terms* $t_0,...,t_7$ (after canonical encoding) and signs bits $A_{s0},...,A_{s7}$, along with 8 8-bit B exponents $B_{e0},...,B_{e7}$, their significands $B_{m0},...,B_{m7}$ (as-is) and their sign bits $B_{s0},...,B_{s7}$ as shown in Figure 4.7.

Block 1 — Exponent: Processing a new set of 8 value pairs starts first at the exponent block. This block adds the A and B exponents in pairs to produce the exponents ABe_i for the corresponding products. A comparator tree takes these product exponents and the exponent of the accumulator and selects the maximum exponent e_{max}. The maximum exponent is used to align all products so that they can be summed correctly. To determine the proper alignment per product the block subtracts all product exponents from e_{max} calculating the alignment offsets δe_i. The maximum exponent is used to also discard terms that will fall out-of-bounds when accumulated. The PE will

skip any terms who fall outside the $e_{max}-12$ range, given that the fractional part of the accumulator is 12b. The block is invoked only *once* per new set of value pairs, before any terms are generated, and regardless of how many terms end up being generated. Accordingly, the minimum effective number of cycles for processing the 8 MACs will be 1 cycle regardless of value (the blocks can be pipelined, and since there are no data dependencies, the pipeline can be kept full). In case one of the resulting products has an exponent larger than the current accumulator exponent, the accumulator will be shifted accordingly prior to accumulation (*acc_shift* signal).

Block 2 — Shift&Reduce: Since multiplication with a term amounts to shifting, this block calculates the number of bits by which each B significand will have to be shifted prior to accumulation. These are the 4-bit terms $K_0,...,K_7$. To calculate K_i we add the product exponent deltas (δe_i) to the corresponding A term t_i. The A sign bits are XORed with their corresponding B sign bits to determine the signs of the products $P_{s0},...,P_{s7}$.

The B significands are complemented according to their corresponding product signs, and then shifted using the offsets $K_0,...,K_7$. The PE uses a shifter per B significand to implement the multiplication. In contrast, a conventional floating point unit would require shifters at the output of the multiplier. Thus *FPRaker* PE effectively completely eliminates the cost of the multipliers.

Bits that are shifted out of the accumulator range from each B operand are rounded using **round-to-nearest-even** (RNE) approach, which is the default rounding scheme for floating-point addition. An adder tree reduces the 8 B operands into a single partial sum as shown in Figure 4.7(2).

Block 3 — Accumulation: The resulting partial sum from step 2 is added to the correctly aligned value of the accumulator register. In each accumulation step, the accumulator register is normalized and rounded using the rounding-to-nearest-even (RNE) scheme. The normalization block updates the accumulator exponent as shown in Figure 4.7(3). When the accumulator value is read out, it is converted to bfloat16 by extracting only 7b for the significand.

In the worst case two K_i offsets may differ by up to 12 since our accumulator has 12 fractional bits. This means that the baseline PE requires relatively large shifters and an accumulator tree that accepts wide inputs. Specifically, the PE requires shifters that can shift up to 12 positions a value that is 8b (7b significant + hidden bit). Had this been integer arithmetic we would need to accumulate $12+8=20b$ wide. However, since this is a floating point unit we will be accumulating only the 14 most significant bits (1b hidden, 12b fractional and the sign). Any bits falling below this range will be included in the sticky bit which is the least significant bit of each input operand.

4.4.1.2 2-Stage Shifting

It is possible to greatly reduce the area cost of the PE by taking advantage of the expected distribution of the exponents. Figure 4.8 shows, for example, the distribution of exponents for a layer of ResNet34. The y-axis reports the frequency each x-axis value appears. For example, a y-axis value of 0.4 for the x-axis value 0 means that the 40% of the exponent values are zero. The vast majority of the exponents of the inputs, the weights and the output gradients lie within a narrow range. This suggests that in the common case the exponent deltas will be relatively small. In addition, the MSBs of the activations are guaranteed to be one (given denormals are not supported [49]). This indicates that very often the $K_0,...,K_7$ offsets would lie within a narrow range. We take advantage of this behavior to reduce the PE area. In our preferred configuration we limit the maximum difference in among the K_i offsets that can be handled in a single cycle to be up to 3. As a result, the shifters need to support shifting by up to 3b and the adder now need to process 12b inputs (1b hidden, $7b+3b$ significant, and the sign bit).

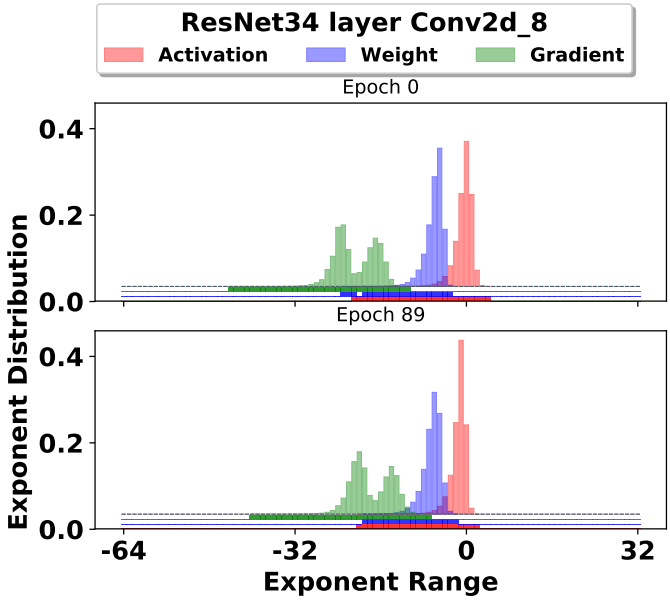

Figure 4.8: Normalized exponent distribution of layer *Conv2d_8* in epochs 0 and 89 of training ResNet34 on ImageNet. The figure shows only the utilized part of the full range [-127:128] of an 8b exponent.

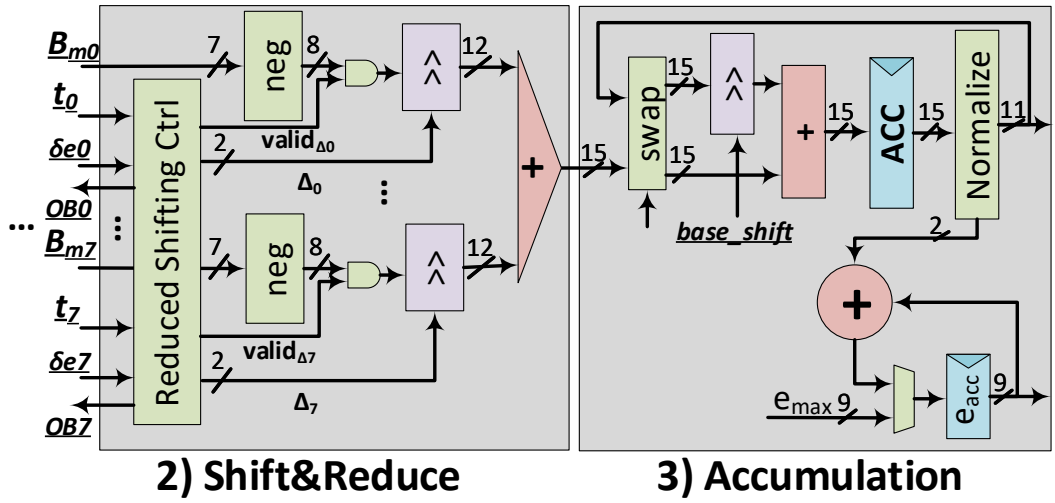

Figure 4.9: *FPRaker* PE: Modified Design.

Figure 4.9 shows the modified *FPRaker* PE where the shifters in the "Shift&Reduce" stage in Figure 4.7 are limited to shift up to 3b. A shared *single* shifter (*base_shift*) after the adder tree serves a dual purpose: First, it aligns the adder tree's output and the accumulator properly. Second, it allows the PE to skip over longer than 3b distances in the input term stream. This is useful, when the next set of terms are at a distance longer than 3b vs. the current ones. Each PE has a control unit (*Reduced Shifting Ctrl*) to generate the modified terms Δ_i and a $valid_{\Delta i}$ signal indicating whether the lane can process its term at the current cycle. The exponent block is not shown for simplicity since it is not affected by the 2-stage shifting optimization. The value of the common shifting stage is

calculated as the minimum of the original shifting stage $(k_0,...,k_7)$ as shown in Figure 4.10 and the amounts of first stage shifting $(\Delta_0,...,\Delta_7)$ are calculated as $\Delta_i = k_i - base\_shift$.

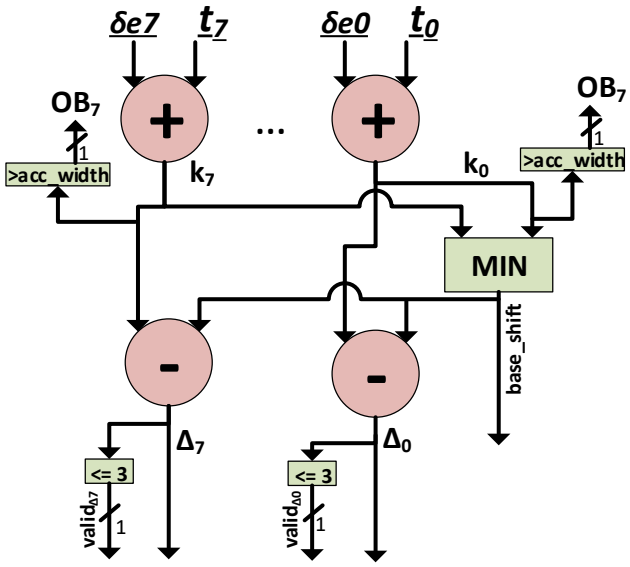

Figure 4.10: *FPRaker* PE control unit (*Reduced Shifting Ctrl*)

4.4.1.3 Skipping out-of-bounds terms

Skipping out-of-bounds terms turns out to be surprisingly inexpensive. The control unit uses a comparator per lane to check if its current K term lies within a threshold with the value of the accumulator precision (comparators are optimized by the synthesis tool for comparing with a constant) as shown in Figure 4.10 and feeds back an OB_i signal to its corresponding term encoder indicating that any subsequent term coming from the same input pair is guaranteed to be ineffectual (out-of-bound) given the current e_{acc} value. Hence, *FPRaker* can boost its performance and energy-efficiency by skipping the processing of the subsequent out-of-bound terms. The OB_i signals of a certain lane across the PEs of the same tile column are synchronized together. The threshold is currently set according to [99] which ensures models converge within 0.5% of the FP32 training accuracy on the ImageNet dataset. However, the threshold can be controlled effectively implementing a dynamic bit-width accumulator that can boost performance by increasing the number of skipped "out-of-bounds" bits, an option we do not investigate further and keep as a potential direction for future work.

4.4.2 Simplified Example

Figure 4.11 shows an example of a simplified PE processing 2 activation-weight pairs each has 5b mantissa including the hidden-bit: A_0=$2^2\times1.1101, B_0$=$2^3\times1.0011$ and A_1=$2^1\times1.1011, B_1$=$2^1\times1.1010$.

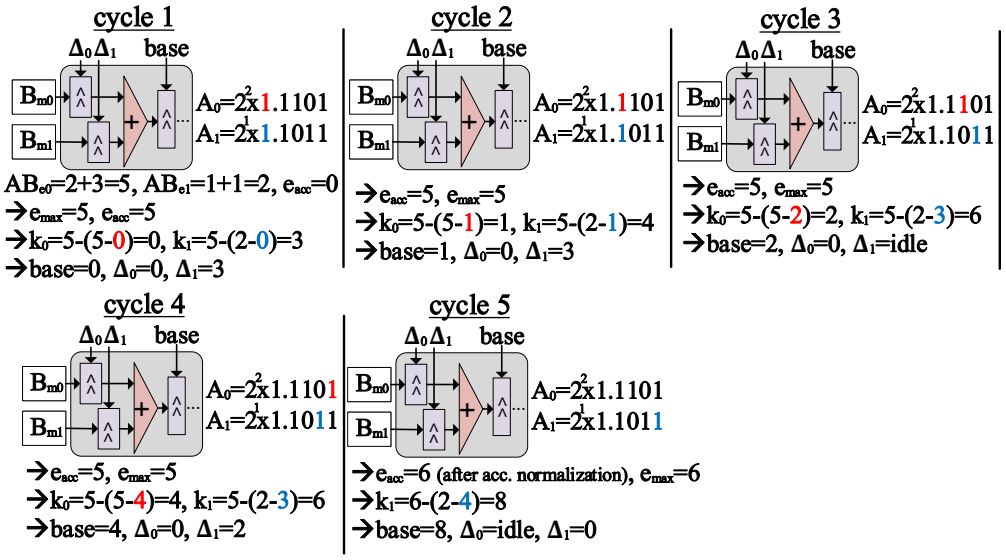

Figure 4.11: *FPRaker* Processing Example

On cycle 1, the exponent block computes the products' exponents AB_{e0} and AB_{e1} (used once per new set of input pairs). Assuming a zero accumulator for simplicity, the *MAX* block computes e_{max}=$max(AB_{e0}, AB_{e1}, e_{acc})$=5. The accumulator exponent is then set to e_{acc}=e_{max}. The absolute terms are computed as k_i=e_{acc}−$(AB_{ei}$−$t_i)$, where t_i are the position of non-zero terms generated by the term encoders shared across the PEs of the same tile column. To limit the PE's shifters range up to 3b, the shared shifter after the adder tree is set to $base$=$min(k_0, k_1)$. The value of the limited shifters are set to Δ_i=k_i−$base$. For cycles 1 & 2, since both Δ_0 and Δ_1 are within 3, both lanes can operate simultaneously. On cycle 3, the difference between k_0 and k_1 is more than 3, which means both terms cannot be processed simultaneously. Hence, lane 1 stalls while lane 0 operates with $base$=k_0 and Δ_0=0. On cycle 4, lane 1 has its term from the previous cycle while lane 0 has a new term. Since the difference between the two terms is within 3, both lanes can operate simultaneously. On cycle 5, lane 0 is idle since it finished its terms while lane 1 processes its final term. To illustrate skipping out-of-bound terms, assume the total precision of the accumulator mantissa is 6b. On cycle 4, lane 1 feeds back a signal to its term encoder indicating that any subsequent term coming from the same input pair is guaranteed to be ineffectual (out-of-bound) term. Hence, lane 1 can skip processing its last term and the PE saves one processing cycle by finishing at cycle 4.

In this example, the *FPRaker* PE has a throughput of 2 $1b$×$5b$ products per cycle. However, an equivalent baseline PE with input 2 lanes can perform 2 $5b$×$5b$ products per cycle. Assuming that we can fit a tile of 5 *FPRaker* PEs within the area of a single baseline PE, *FPRaker* will have a throughput of 5 2×$1b$×$5b$ products equivalent to the baseline. With the presence of a large number of zero and out-of-bound terms, *FPRaker* can have a performance speedup over the baseline.

4.4.3 Sharing the Exponent Block

In the common case, processing a group of A values will require multiple cycles since some of them will be converted into multiple terms. During that time the inputs to the exponent block will not change. To further reduce area we can take advantage of this expected behavior and share the exponent block across multiple PEs. The decision of how many PEs to share an exponent block over can be based on the expected bit-sparsity. The lower the bit-sparsity the higher the processing time per PE and the less often it will need a new set of exponents. Hence,

more PEs can share the exponent block. For the studied models sharing one exponent block per two PEs proved best. Figure 4.12 shows the modified design. The unit as a whole accepts as input one set of 8 A inputs and two sets of B inputs, B and B'. The exponent block can process one of (A,B) or (A,B') at a time. During the cycle when it processes (A,B) the multiplexer for PE#1 passes on the e_{max} and exponent deltas directly to the PE. Simultaneously, these values will be latched into the registers in front of the PE so that they remain constant while the PE processes all terms of input A. When the exponent block processes (A,B') the aforementioned process proceeds with PE#2. With this arrangement both PEs must finish processing all A terms before they can proceed to process another set of A values. Since the exponent block is shared, each set of 8 A values will take at least 2 cycles to be processed (even if it contains zero terms).

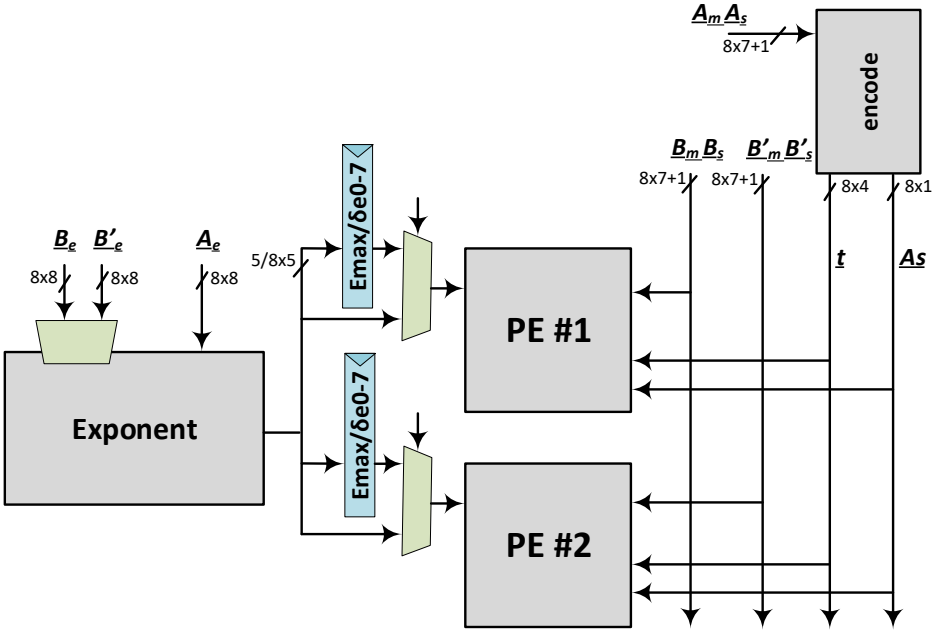

Figure 4.12: Reducing area by sharing the exponent block between two PEs.

4.4.4 Tile Organization

By utilizing per PE buffers it is possible to exploit data reuse temporally. To exploit data reuse spatially we can arrange several PEs into a tile. Figure 4.13 shows an example of a 2×2 tile of PEs and each PE performs 8 MAC operations in parallel. The PEs along the same column share the same input activations, while the PEs along the same row share their weight inputs. Each column has a shared booth encoder which feeds the effectual terms to the constituent PEs. Each pair of PEs per column shares an exponent block as previously described. The B and B' inputs are shared across PEs in the same row. For example, during the forward pass we can have different filters being processed by each row and different windows processed across the columns. Since the B and B' inputs are shared all columns would have to wait for the column with the most Ai terms to finish before advancing to the next set of B and B' inputs. To reduce these stalls the tile introduces per B and B' buffers. By having N such buffers per PE allows the columns be at most N sets of values ahead.

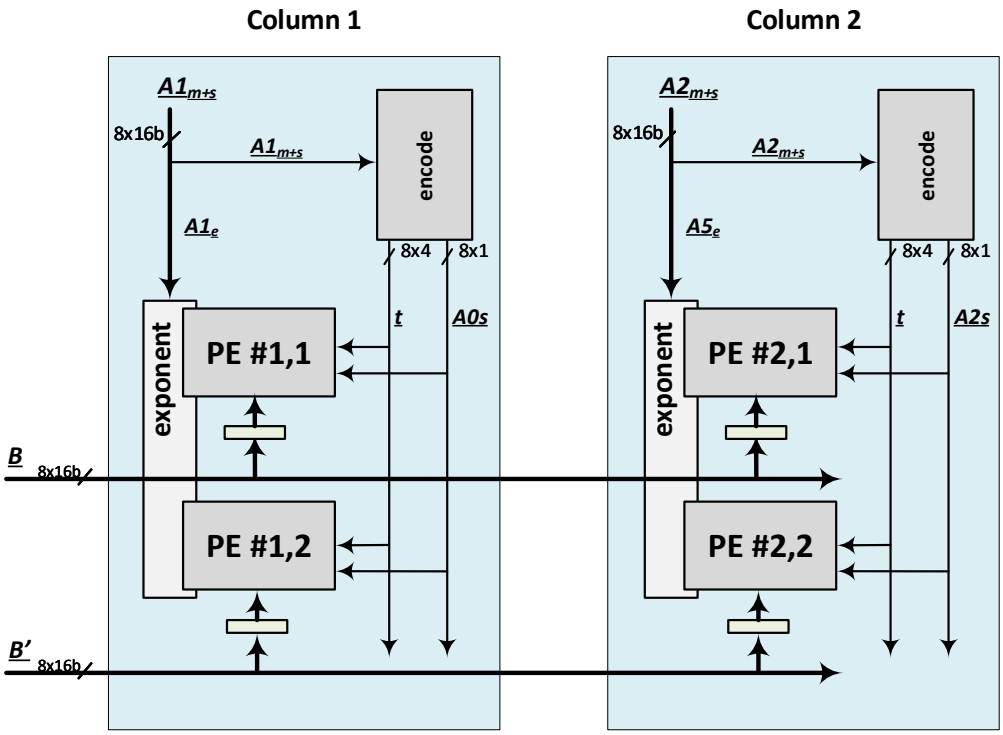

Figure 4.13: A 2×2 PE *FPRaker* Tile.

4.5 Exponent Base-Delta Compression

Motivated by the narrow value distribution shown in Figure 4.8, we studied the spatial correlation of values during training. We found that consecutive values across all dimenions (channel, H, or W) have similar values. This is true for the activations, the weights and the output gradients. Similar values in floating-point have similar exponents, a property which we can exploit through a base-delta compression scheme [90]. In our experiments, we block values channel-wise (we present evidence, however, that the value corellation persists along the H dimension as well) into groups of 32 values each, where the exponent of the first value in the group is the base (B) and we compute the delta exponent (ΔE) for the rest of the values in the group relative to it. The precision (P) of the delta exponents is dynamically determined per group and is set to the maximum precision of the resulting delta exponents per group similar to the approach of Delmas et al. [67]. The delta exponent precision (3b) and the base exponent are attached to the header of each group as metadata. Figure 4.14 shows the compression and decompression units.

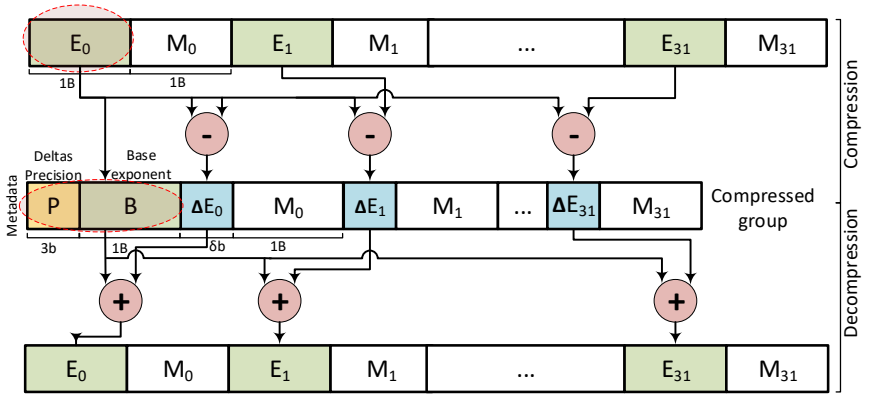

Figure 4.14: Exponent base-delta compression/decompression for a group of 32 values

Figure 4.15 shows the normalized exponent footprint after the base-delta compression. Our technique is effective for both channel-wise and spatial (H dimension shown) dataflows. On average, the normalized exponent footprint is 31.9%, 40.1% and 42.2% for the activations, weights and gradients, respectively. We use this compression scheme to reduce the off-chip memory bandwidth. Values are compressed at the output of each layer and before writing them off-chip, and they are decompressed when they are read back on-chip.

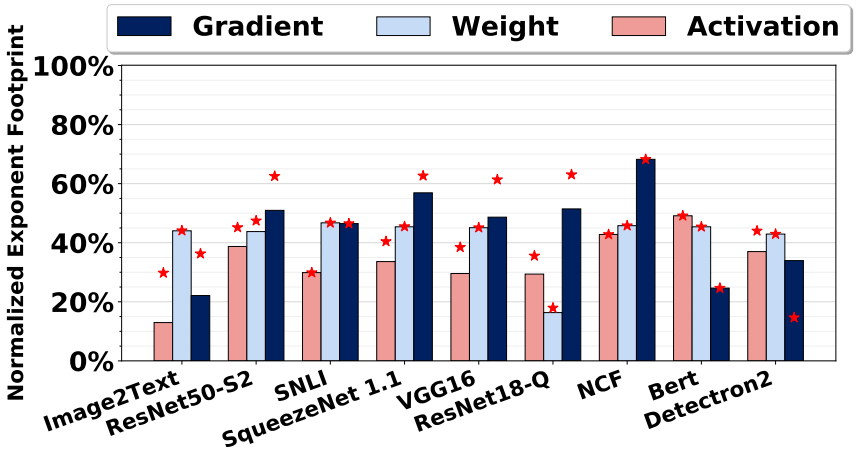

Figure 4.15: Memory savings due to exponent base-delta compression. Bars and markers represent compression channel-wise and spatial-wise, respectively.

4.6 Data Supply

Focusing solely on computation is insufficient. Data transfers account for a significant portion and often dominate energy consumption in deep learning. Accordingly, it is essential to consider what the memory hierarchy needs to do to keep the execution units busy. A challenge with training is that while it processes three arrays I, W and G the order in which their elements are grouped differs across the three major computations (Eq. 4.1 through 4.3). However it is possible to rearrange the arrays as they are read from off-chip. For this purpose we store the arrays in memory using a container of "square" of 32×32 bfloat16 values. This a size that matches well the typical row sizes of DDR4 memories and allows us to achieve high bandwidth when reading values from off-chip. A container includes values from coordinates (c,r,k) (channel, row, column) to $(c+31,r,k+31)$ where c and k are divisible by 32 (padding is used as necessary). Containers are stored in channel, column, row order. When read from off-chip memory, the container values are stored in the exact same order in the multi-banked on-chip buffers. The tiles can then access data directly reading 8 bfloat16 values per access. The weights and the activation gradients however need to be processed in different order depending on the operation performed. Effectively, the respective arrays must be accessed in the transpose order during one of the operations. For this purpose we incorporate transposer units on-chip. A transposer reads in 8 blocks of 8 bfloat16 values from the on-chip memories. The transposer then can provide 8 blocks of 8 values each composed of a single value from each of the 8 original blocks read from memory effectively transposing the tensor. Collectively these blocks form an 8×8 block of values. The transposer can read out 8 blocks of 8 values each and send those to the processing units. Each of these blocks however is read out as a column from its internal buffer. This effectively transposes the 8×8 value group.

4.7 Evaluation

This section evaluates *FPRaker* against an equivalent baseline architecture that uses bit-parallel floating-point MAC units. First, we present the experimental methodology used in this thesis. Next, we discuss the post-layout area results and accordingly configure both *FPRaker* and baseline accelerators for *ISO-compute-area* comparison. We then show the performance and energy-efficiency of *FPRaker* over the baseline, followed by an extensive performance analysis study. Afterwards, we present an accuracy study indicating that *FPRaker* processing does not affect final training accuracy. We conclude this section by showing the performance of *FPRaker* with a technique that profiles the accumulation bit-width per layer during training.

4.7.1 Methodology

A custom trace-based cycle-accurate simulator was developed in C++ to model the execution time of *FPRaker* and of the baseline architecture. The simulator is trace-driven using the traces of the forward and backward passes collected in each training epoch while training the models on a NVIDIA RTX 2080 Ti GPU. Besides modeling timing behavior of the simulator also models value transfers and computation in time faithfully and checks the produced values for correctness against the golden values. The simulator was validated with microbenchmarking where for each computation phase of training ($A \times W$, $A \times G$, $G \times W$), we validate the output of the simulator with its corresponding collected output traces. For area and power analysis, both *FPRaker* and the baseline designs were implemented in Verilog and synthesized using Synopsys' Design Compiler [66] for a 600 MHz clock frequency with a 65nm TSMC technology (due to licensing restrictions we cannot get access to a better technology). We use Cadence Innovus [12] for layout generation. We use Intel's PSG ModelSim [55] to generate data-driven activity factors we feed to Innovus to estimate the power. The baseline MAC unit was optimized for area, energy and latency [33]. We use an efficient bit-parallel fused MAC unit as the baseline PE. The constituent multipliers are both area and latency efficient, and are taken from the DesignWare IP library developed by Synopsys. Further, we optimize the baseline units for deep learning training by reducing the precision of its I/O operands to bfloat16 and accumulating in reduced precision with chunk-based accumulation similar to FPRaker units. The area and energy consumption of the on-chip SRAM Global Buffer (GB) is divided into activation, weight, and gradient memories that were modeled using CACTI [50]. The Global Buffer has an odd number of banks to reduce bank conflicts for layers with a stride greater than one. To estimate the latency and energy consumption of the off-chip DRAM memory we use the model provided by Micron [81]. Following an iso-compute-area comparison, since the area of the *FPRaker* tile is 4.5× smaller than the baseline, we configured both *FPRaker* and the baseline are shown in Table 4.1.

Table 4.1: Baseline and *FPRaker* configurations.

	FPRaker	**Baseline**
Tile Configuration	8×8	8×8
Tiles	36	8
Total PEs	2304	512
Multipliers/PE	8	8 BFLOAT16
MACs/cycle	-	4096
Scratchpads	2KB each	
Global Buffer	4MB × 9 banks	
Off-chip DRAM Memory	16GB 4-channel LPDDR4-3200	

To evaluate our accelerator, we collected traces for one random mini-batch during the forward and backward

pass in each epoch of training. All models were trained long enough to attain the maximum top-1 accuracy as reported by the original authors. To collect the traces, we trained each model on a NVIDIA RTX 2080 Ti GPU and stored all of the inputs and outputs for each layer using Pytorch Forward and Backward hooks. For BERT we traced BERT-base and the fine-tuning training for a GLUE task. The simulator uses the traces to model execution time and collects activity statistics so that energy can be modeled.

4.7.1.1 Comparison under ISO-Compute-Area Constraints

Since *FPRaker* processes one of the inputs term-serially, a single *FPRaker* processing engine can never outperform a conventional PE that processes the same number of inputs. *FPRaker* relies on parallelism to extract more performance. This is only possible if we can afford to use more *FPRaker* units than conventional units.

One approach is to use an iso-compute area constraint. This is constraint that has been popular when comparing different accelerators [87, 102, 104]. With the *iso-compute-area* comparison method, we constrain the proposed accelerator to use the same on-chip compute area as the baseline accelerator. Particularly, given a certain silicon area budget for the compute logic, we study the performance and energy-efficiency of our proposed accelerator compared to the baseline accelerator. This method is widely used in the computer architecture community due to its simplicity. Accordingly, we first measure the area of our units so that we can then determine appropriate configurations under the iso-compute-area constraint.

In our case, we are interested to determine how many *FPRaker* tiles we can fit in the same area for a baseline tile. For this we take into account only the compute cores as associated logic and not the scratchpads.

4.7.2 Area

We configure the *FPRaker* tile similar to the baseline as discussed previously in Section 4.2. Post layout, and taking into account only the compute area, an *FPRaker* tile occupies 22% the area vs. the baseline tile. Table 4.2 reports the corresponding area and power per tile. Accordingly, to perform an iso-compute-area comparison, we configure the baseline accelerator to have 8 tiles and *FPRaker* with 36 tiles as shown previously in Table 4.1 The area for the on-chip SRAM global buffer is $344mm^2$, $93.6mm^2$, and $334mm^2$ for the activations, weights, and gradients, respectively. Both *FPRaker* and the baseline have the same on-chip memory capacity.

Table 4.2: Breakdown of the area and power consumption per tile of *FPRaker* vs. Baseline.

Area [μm^2]				
	PE Array	**Term Encoders**	**Total**	**Normalized**
FPRaker	304,118	12,950	317,068	0.22×
Baseline	1,421,579	N/A	1,421,579	1×
Power [mW]				
FPRaker	104	5.5	109.5	0.23×
Baseline	475	N/A	475	1×
Energy Efficiency Normalized to Baseline				
FPRaker			1.4×	

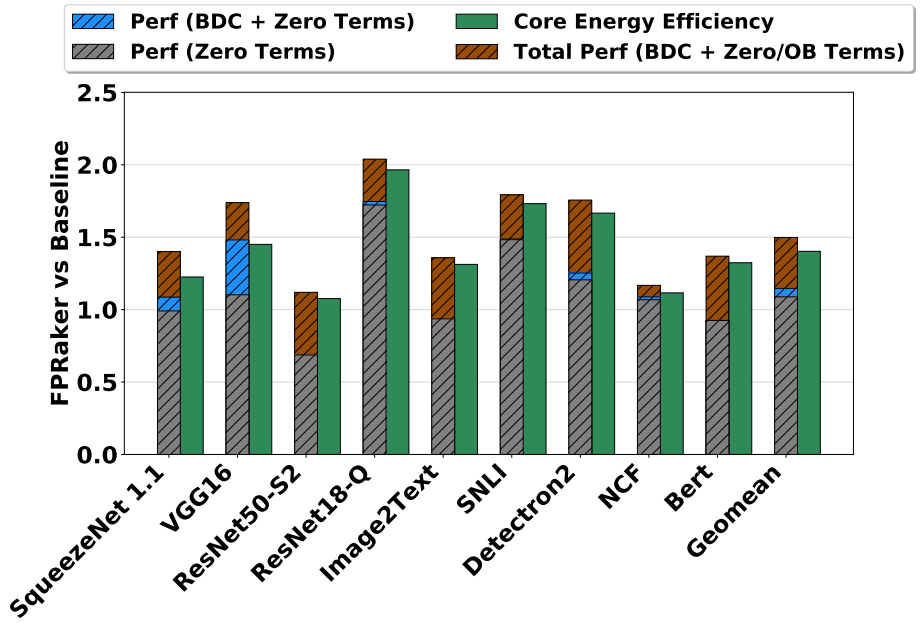

Figure 4.16: ISO-compute-area performance and energy-efficiency comparison between *FPRaker* and the baseline.

4.7.3 Execution Time

Figure 4.16 shows the performance breakdown of *FPRaker* due to the base-delta compression (BDC), and skipping zero and out-of-bound (OB) terms relative to the baseline. On average, *FPRaker* outperforms the baseline by 1.5× (skipping zero terms: 9%, BDC: 5.8%, skipping out-of-bound terms: 35.2%). From the studied convolution-based models, ResNet18-Q benefits the most from *FPRaker* where the performance improves by 2.04× over the baseline. Training for this network incorporates PACT quantization and as a result most of the activations and weights throughout the training process can fit in 4b or less. This translates into high term sparsity which *FPRaker* exploits. This result demonstrates that *FPRaker* can deliver benefits with specialized quantization methods without requiring that the hardware be also specialized for this purpose.

SNLI, NCF, and Bert are dominated by fully connected layers. While in fully-connected layers there is no weight reuse among different output activations, training can take advantage of batching to maximize weight reuse across multiple inputs (e.g., words) of the same input sentence which results in higher utilization of the tile PEs. Speedups follow bit-sparsity. For example, *FPRaker* achieves a speedup of 1.8× over the baseline for SNLI due its high bit-sparsity.

4.7.4 Energy Efficiency

Figure 4.16 shows the total energy efficiency of *FPRaker* over the baseline architecture for each of the studied models. On average, *FPRaker* is 1.4× more energy efficient compared to the baseline considering only the compute logic and 1.36× more energy efficient when on- and off-chip memories are taken into account. The energy-efficiency improvements follow closely the performance benefits. The benefits are higher at around 1.7× for SNLI and Detectron2. The quantization in ResNet18-Q boosts the compute logic energy efficiency to as high as 1.97×. Figure 4.17 shows the energy consumed by *FPRaker* normalized to the baseline as a breakdown across three main components: compute logic, off-chip and on-chip data transfers. Figure 4.17 further breaks down

the *FPRaker* core into "Compute" (PE stages 1 and 2), "Control" (PE control units and shared term encoders), and "Accumulation" (PE stage 3). *FPRaker* along with the exponent base-delta compression reduce the energy consumption of the compute logic and off-chip memory significantly.

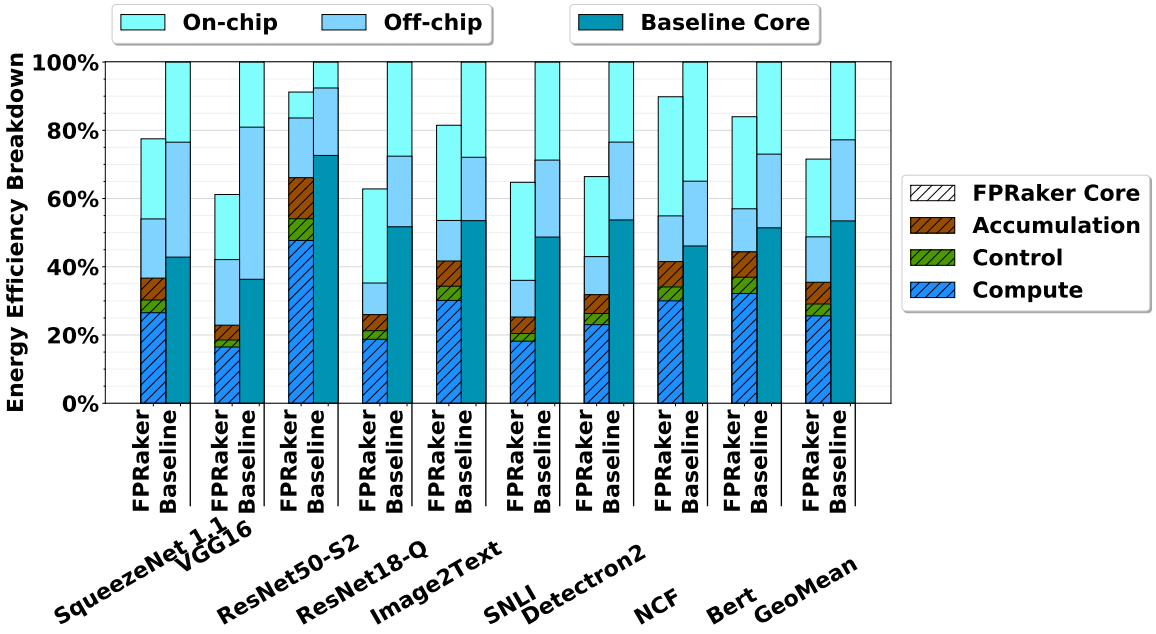

Figure 4.17: Overall Energy Efficiency of *FPRaker* vs Baseline.

4.7.5 Performance Analysis

We conducted a performance analysis study to detect where speedup comes from, breakdown *FPRaker*'s execution time and detect the performance overheads per processing element, and study how the tile configuration can affect performance.

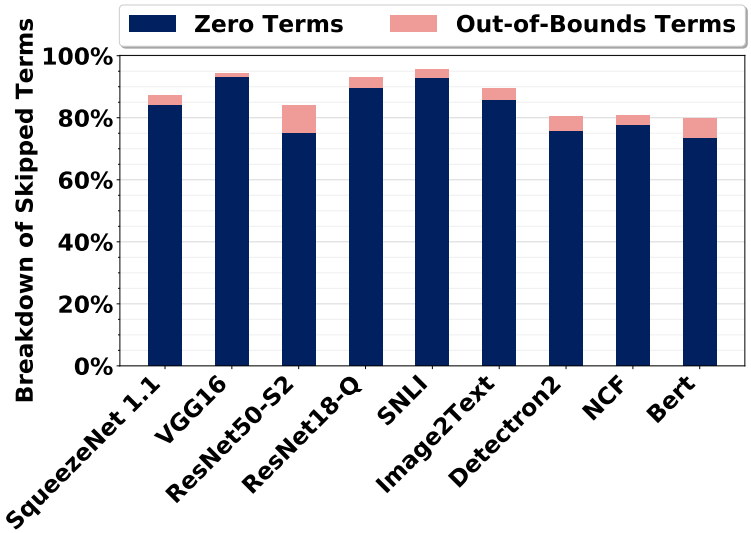

Figure 4.18: Breakdown of skipped terms by *FPRaker*.

4.7.5.1 Skipped Terms

Figure 4.18 shows a breakdown of the terms *FPRaker* skips. There are two cases: 1) skipping zero terms, and 2) skipping non-zero terms that are out-of-bounds due to the limited precision of the floating-point representation.

Skipping out-of-bounds terms increases term sparsity for ResNet50-S2 and Detectron2 by around 10% and 5.1%, respectively. Networks with high sparsity (zero values) such as VGG16 and SNLI benefit the least from skipping out-of-bounds terms with the majority of term sparsity coming from zero terms. This is because there are few terms to start with. For ResNet18-Q, most benefits come from skipping zero terms as the activations and weights are effectively quantized to 4b values. However, as Figure 4.16 showed, skipping out-of-bound terms improves performance much more than the fraction of terms skipped would suggest. Recall, that all lanes must wait for the slowest one to finish processing amplifying the effect on performance across all lanes. In the worse case, all other 7 lanes are waiting for a single one to finish, wasting 7 execution lanes. Skipping out-of-bound terms reduces this synchronization overhead.

4.7.5.2 Computation Phase

Figure 4.19 reports speedup for each of the 3 phases of training: the $A \times W$ in forward propagation, and the $A \times G$ and the $G \times W$ to calculate the weight and input gradients in the backpropagation, respectively. *FPRaker* consistently outperforms the baseline for all three phases. The speedup depends on the amount of term sparsity, and the value distribution of A, W, and G across models, layers, and training phases. The less number of terms a value has, the higher the potential for *FPRaker* to improve performance. However, due to the limited shifting that the *FPRaker* PE can perform per cycle (up to 3 positions) how terms are distributed within a value impacts the number of cycles needed to process it. This behavior applies across lanes to the same PE and across PEs in the same tile. In general, the set of values that are processed concurrently will translate into a specific term sparsity pattern. *FPRaker* favors patterns where the terms are close to each other numerically.

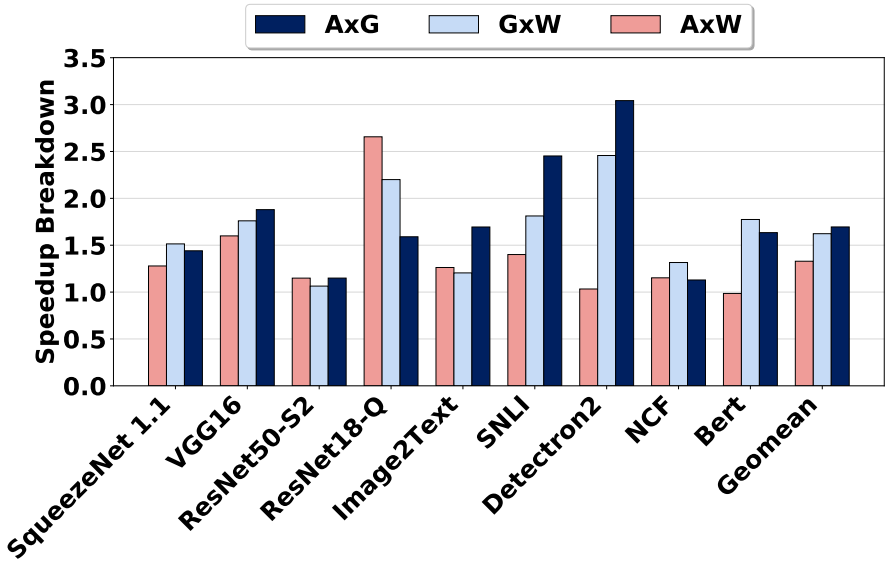

Figure 4.19: Breakdown of *FPRaker* speedup over the baseline.

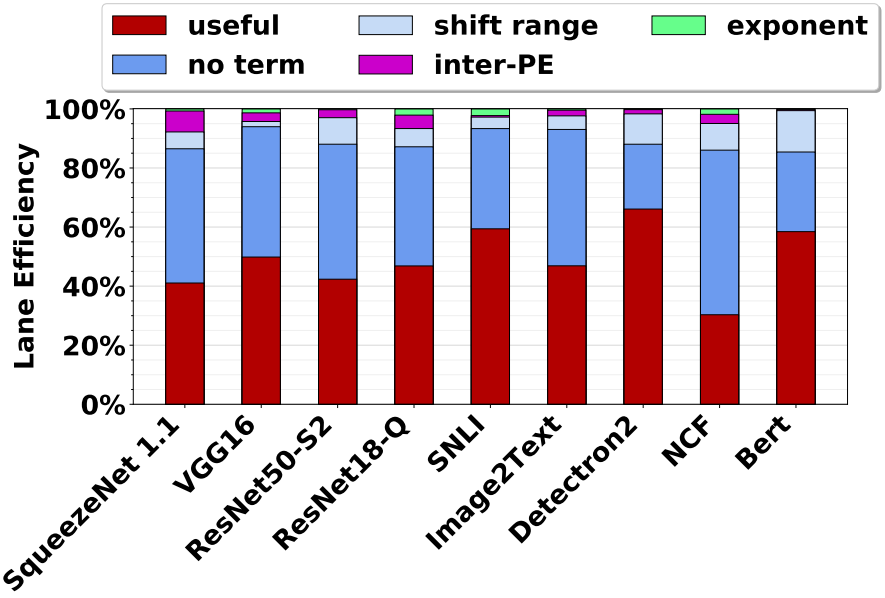

Figure 4.20: Breakdown of execution cycles of *FPRaker*.

4.7.5.3 Where Cycles Go

Figure 4.20 reports a breakdown of PE lane utilization and highlights performance bottlenecks. There are several reasons why stalls occur: 1) inter-PE synchronization, 2) intra-PE synchronization, and 3) sharing the exponent block. Intra-PE synchronization stalls can happen in two scenarios: a) imbalance in the number of terms assigned for each lane within a PE due to uneven distribution of the term sparsity in the model which results in idle lanes waiting for the slowest lane to finish execution ("no terms"), b) high span across consecutive terms within a lane which cause stalls due to the limited per cycle shift range of the PE ("shift range").

Stalls due to sharing the exponent block are rare. The more bit-sparsity a model has the higher pressure on the exponent block and the higher the chance that it will not be able to keep up. Exponent stalls are noticeable for ResNet18-Q since the values there are effectively 4b. We can see a similar behavior for SNLI due to its high bit-sparsity. However, there are other types of stalls that reduce pressure on the shared exponent block. First are stalls due to the limited per cycle shifting ability of the PEs. These are relatively few thus demonstrating that this technique presents a good performance vs. area trade-off. Second are stalls due to cross-lane term imbalance. These are the highest cause of PE underutilization; 32.8% on average, and at most 55% for NCF. Term imbalance is lowered if we reduce the number of lanes per PE or if we add weight buffers to allow faster lanes to proceed with the next set of weights albeit with a higher area overhead. However, doing so would increase the cost of the PE. This investigation is left for future work. Third are stalls due to inter-PE synchronization which are also rare. The ability for PE columns to run ahead by just one set sufficiently hides these stalls.

Figure 4.21 shows the benefit of skipping out-of-bound terms in reducing the synchronization overheads (an average of 30.3% overall reduction) by improving the load balancing across the PE lanes.

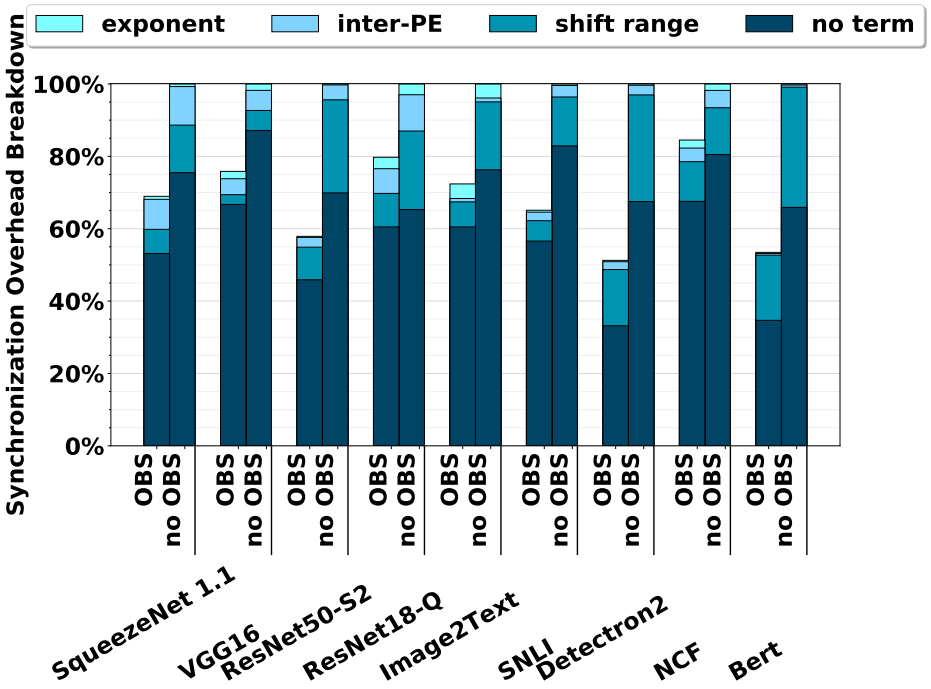

Figure 4.21: Effect of out-of-bound terms skipping (OBS) on the synchronization overhead.

4.7.5.4 Performance Over Time

Figure 4.22 shows the speedup of *FPRaker* over the baseline over time and throughout the training process for all the studied networks. The measurements show three different trends. For VGG16 speedup is higher for the first 30 epochs after which it declines by around 15% and plateaus. For ResNet18-Q, the speedup increases after epoch 30 by around 12.5% and stabilizes. This can be attributed to the PACT clipping hyperparameter being optimized to quantize activations and weights within 4-bits or below. For the rest of the networks, speedups remain stable throughout the training process. Overall, the measurements show that performance of *FPRaker* is robust and that it delivers performance improvements across all training epochs.

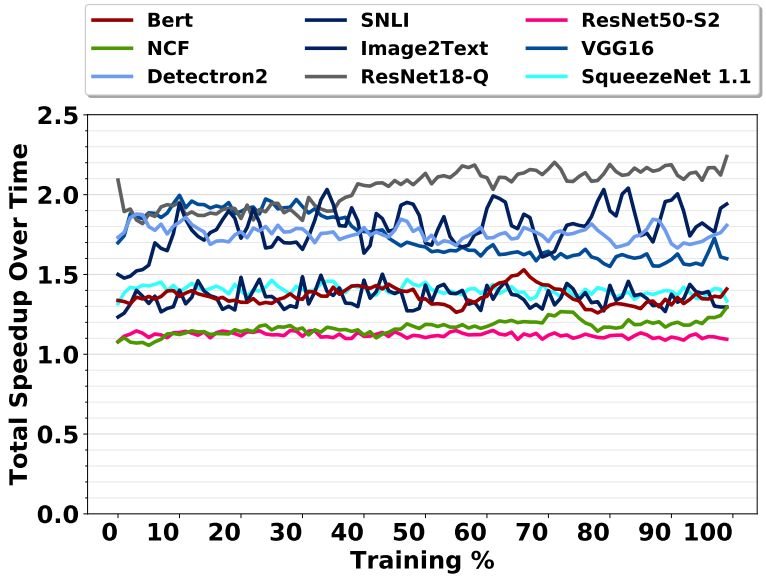

Figure 4.22: Speedup of *FPRaker* vs. the baseline over time.

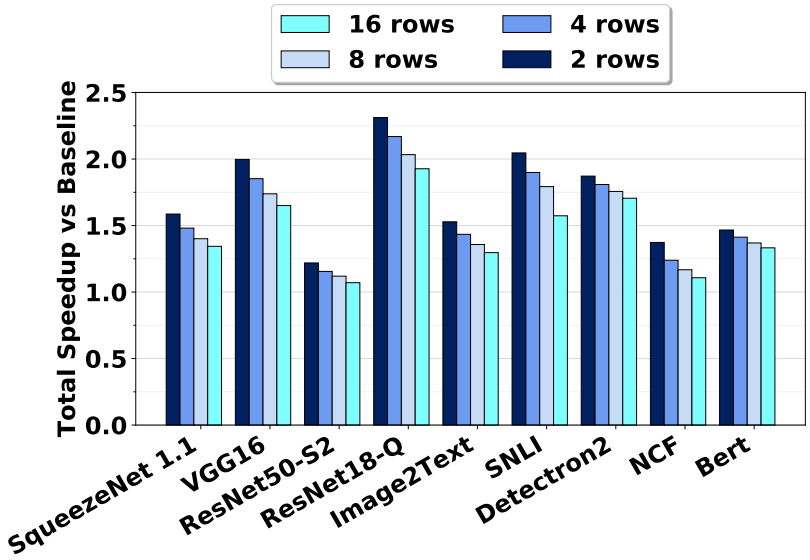

Figure 4.23: Speedup of *FPRaker* vs. the baseline with varying the number of rows per tile.

4.7.5.5 Effect of Tile Organization

As shown in Figure 4.23, doubling the number of rows per tile reduces performance by 6% on average. This reduction in performance is due to synchronization among a larger number of PEs per column. When the number of rows increases, more PEs share the same set of A values. An A value that has more terms than the others will now affect a larger number of PEs which will have to wait to finish processing. Since each PE processes a different combination of input vectors, each can be affected differently by intra-PE stalls such as "no term" stalls or "shift range" stalls. However, there is a trade-off between improving the energy-efficiency through higher spatial-reuse of data with increasing the number of rows per tile and the increased synchronization overhead. Figure 4.24 shows a breakdown of where time goes in each configuration. It can be seen that the stalls for the inter-PE synchronization increase and so do those for stalling for other lanes ("no term").

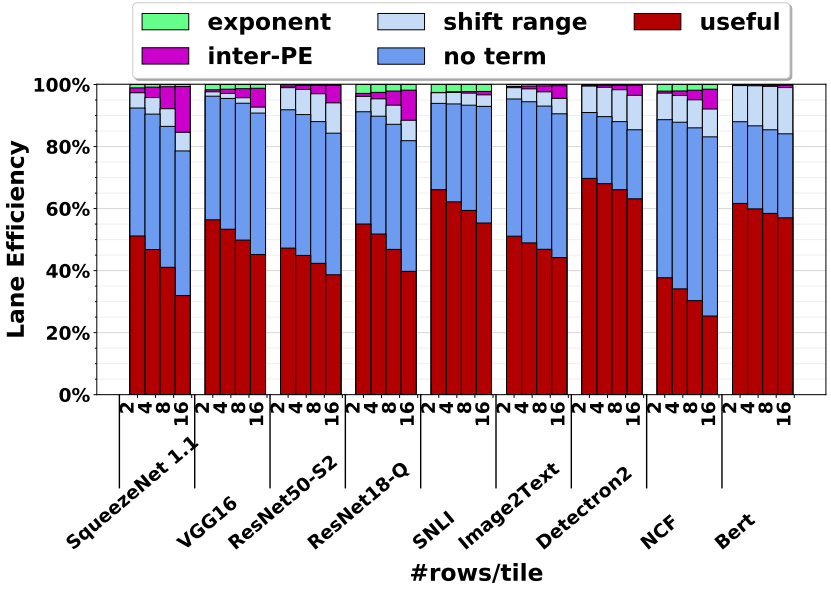

Figure 4.24: Varying the number of rows: Breakdown of Cycles.

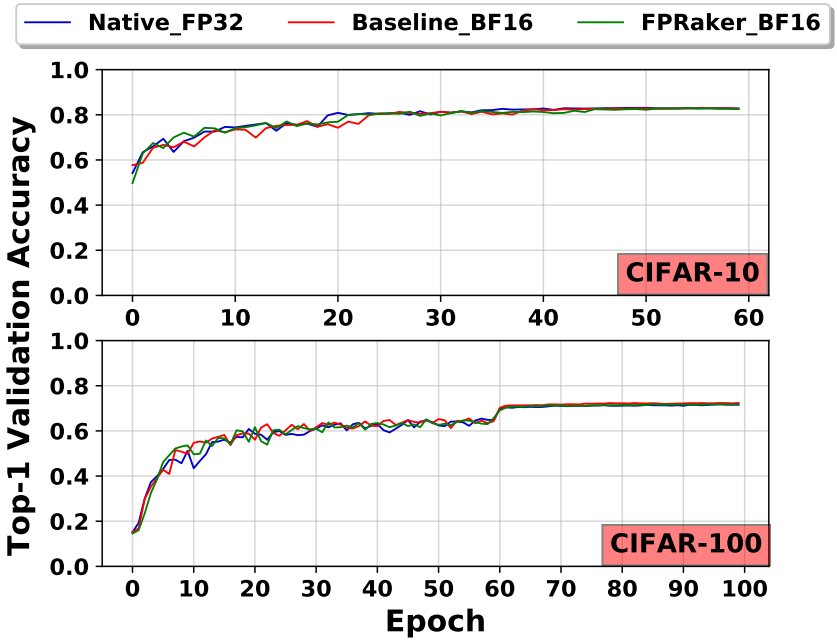

Figure 4.25: Top-1 validation accuracy of training ResNet18 by emulating the FPRaker processing in PlaidML.

4.7.6 Accuracy Study

To study the effect of training with FPRaker on accuracy, we emulated the bit-serial processing of FPRaker PE during end-to-end training in PlaidML [92] which is a machine learning framework based on an OpenCL compiler at the backend. We force PlaidML to use the mad() function for every multiply-add during training. We override the mad() function with our implementation to emulate the processing of the FPRaker PE. We trained ResNet18 on CIFAR-10 and CIFAR-100 datasets as shown in Figure 4.25. The blue line shows the top-1 validation accuracy for training natively in PlaidML with FP32 precision. The baseline performs bit-parallel MAC with I/O operands precision in bfloat16 format which is known to converge and is supported in the industry, e.g. in Google's TPU. The figure shows that both the baseline and FPRaker emulated versions converge at epoch 60 for both datasets with accuracy difference within 0.1% relative to the native training version. This is expected since FPRaker skips only ineffectual work, i.e., work which does not affect final result in the baseline MAC processing.

4.7.7 Per Layer Accumulator Width Profiling

Conventionally, training uses bfloat16 for all computations. As we noted in the introduction, there have been proposals for using mixed datatype arithmetic where some of the computations used fixed-point instead [23, 28, 80, 86]. Sakr et. al [98], propose to use floating-point where the number of bits used by the mantissa varies per operation and per layer. We use the suggested mantissa precisions while training AlexNet and ResNet18 on ImageNet. Figure 4.26 shows the performance of *FPRaker* following this approach. *FPRaker* can dynamically take advantage of the variable accumulator width per layer to skip the ineffectual terms mapping outside the accumulator boosting overall performance. Training ResNet18 on ImageNet with per layer profiled accumulator width boosts the speedup of *FPRaker* by 1.51×, 1.45× and 1.22× for $A \times W$, $G \times W$ and $A \times G$, respectively achieving an overall speedup of 1.56× over the bit-parallel baseline accelerator, compared to a 1.13× speedup that is achieved by training *FPRaker* with a fixed accumulator width. Adjusting the mantissa length while using a bfloat16 container manifests itself a suffix of zero bits in the mantissa.

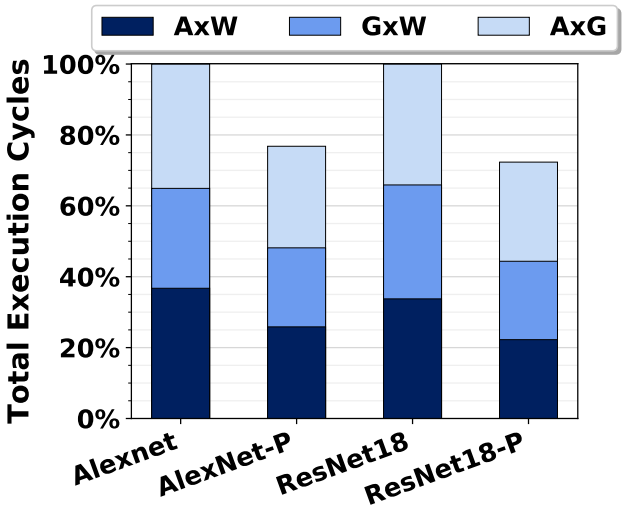

Figure 4.26: Performance of *FPRaker* with per layer profiled accumulator width [98] vs. fixed accumulator width.

4.8 Summary

In this chapter, we showed that bit-sparsity is significantly high during training, and the average potential speedup for exploiting bit-sparsity in one of the MAC operands is 6.5× for the networks studied. We presented *FPRaker*, a hardware accelerator for training DNNs that takes advantage of the inherent bit-sparsity during training to boost performance and energy-efficiency. *FPRaker* is designed to skip the ineffectual computations in both convolution and fully-connected layers. By decomposing the multiplication into a series of "shift-and-add" operations, *FPRaker* can skip fine-grain ineffectual computations due to zero terms in the serialized operand, or non-zero terms with partial sums mapping outside the range of the accumulator (*out-of-bounds* terms). We showed that *FPRaker* benefits from advances in training-time pruning, quantization, or accumulator bit-width profiling. We found that for the studied networks, *FPRaker* is on average 1.5× faster and 1.4× more energy-efficient. Further, we proposed a memory compression technique for the exponent part of the values during training based on base-delta compression that takes advantage of the narrow value distribution during training to reduce the off-chip memory bandwidth.

Chapter 5

Conclusion and Future Work

5.1 Summary of Contributions

In summary, this thesis makes the following contributions:

- We demonstrate that a large fraction of the work performed during training is ineffectual. To expose this ineffectual work we decompose each multiplication into a series of single bit multiply-accumulate operations. This reveals two sources of ineffectual work: First, more than 85% of the computations are ineffectual since one of the inputs is zero. Second, the combination of the high dynamic range (exponent) and the limited precision (mantissa) often yields values which are non-zero, yet too small to affect the accumulated result, even when using extended precision (e.g., trying to accumulate 2^{-64} into 2^{64}). This observation led us to consider whether it is possible to use *bit-skipping* (bit-serial where we skip over zero bits) processing to exploit these two behaviors.
- We propose *FPRaker*, a processing tile for *training* accelerators which exploits both bit-sparsity and out-of-bounds computations. *FPRaker* comprises several adder-tree based processing elements organized in a grid so that it can exploit data reuse both spatially and temporally. The processing elements multiply multiple value pairs concurrently and accumulate their products into an output accumulator. They process one of the input operands per multiplication as a series of signed powers of two hitherto referred to as *terms*. The conversion of that operand into powers of two is performed on the fly; all operands are stored in floating point format in memory. The processing elements take advantage of ineffectual work that stems either from mantissa bits that were zero or from out-of-bounds multiplications given the current accumulator value. The tile is designed for area efficiency and it incorporates the following design choices for this purpose: a) The processing element limits the range of powers-of-two that they can be processed simultaneously greatly reducing the cost of its shift-and-add components. b) A common exponent processing unit that is time-multiplexed among multiple processing elements. c) The power-of-two encoders are shared along the rows. d) Per processing element buffers reduce the effects of work imbalance across the processing elements. e) The PE implements a low cost mechanism for eliminating out-of-bound intermediate values. Skipping out-of-bound intermediate values not only reduces the amount of work, but more importantly proves very effective in reducing the effect of cross-lane synchronization.
- We propose a simple and low-overhead memory compression technique for the exponent part of floating-point values during training. We observe that values typically have narrow distribution during training where consecutive values across the tensor channels have similar values and hence their exponents. Accordingly, we

encode exponents for each group of values using the base-delta compression scheme. We use this encoding to save off-chip memory bandwidth when reading and storing data to the off-chip DRAM.

5.2 Directions for Future Work

The pervasive applications of deep learning and the end of Dennard scaling have been driving efforts for accelerating deep learning inference and training. This thesis studied the data characteristics during training DNNs and proposed a hardware accelerator and a memory compression scheme to reduce the amount of computation and off-chip memory transfers, respectively. This section presents some potential research directions for future work in the area of deep learning hardware acceleration.

This thesis evaluated *FPRaker* for training, however it can naturally also be used for inference. While many neural network models can use fixed-point arithmetic there are models that still require floating-point arithmetic. For example, these include models that process natural language or recommendation systems. Whether *FPRaker* will provide benefits for inference with such models needs further attention.

Interestingly, it has been shown that training can be enhanced by the presence of noise [110] and reduced precision has been represented as a form of regularization [20]. This indicates that training with approximate/reduced precision computing may offer better performance-accuracy tradeoffs to training. While training typically needs high precision to converge to the state-of-the-art accuracy, dynamically-controlled intermittent approximation/quantization may prove benficial.

As shown in Section 4.7.5.3, *FPRaker* loses 38.6% of its potential speedup due to intra-PE synchronization overhead, which results due to work imbalance among the lanes of a PE where some lanes have more terms to process leading to stalling the PE till the slowest lane finishes processing. We showed that skipping out-of-bound terms can significantly reduce the synchronization overheads and hence boosting performance with an inexpensive area overhead. To further alleviate the synchronization overheads on the hardware-level, adding an extra input buffer per tile can allow breaking the synchronization across the PE lanes, i.e., faster lanes do not have to wait for slower lanes and can advance to process the next pair of inputs, albeit with an additional area and control complexity overhead. Reducing the input MAC operands can reduce the intra-PE synchronization, however it also reduces the data parallelism.

Multiple approaches can implemented on the software-level to reduce the synchronization overhead. Conventionally, training DNNs on graphics processing units (GPUs) or bit-parallel accelerators is performed by selecting a fixed high precision format for the end-to-end training, which is typically FP32 or Bfloat16 format. However, this approach amounts to worst-case design. Ideally, precision would be optimized per layer, training epoch, and even the computation phase ($A \times W$, $G \times W$, $A \times G$) of a model. For example, a different precision can be assigned to each layer during training depending on the layer's sensitivity to quantization. Further, a model might require less precision during early training epochs while requiring gradually increasing precision as it gets closer to convergence for fine tuning. Also, it was shown that gradients have wider value distribution compared to activations or weights, and hence require higher precision. Accordingly, computations in the forward pass ($A \times W$) can be performed with less precision than computations in the backward pass ($G \times W$, $A \times G$). Reducing the precision of the MAC operations increases the number of zero terms. Similarly, the accumulator precision can be dynamically assigned during training on a fine-granularity, and not just profiled per layer as discussed in Section 4.7.7. Reducing the accumulator precision increases the number of out-of-bounds terms. Since *FPRaker* can adapt dynamically to different precisions, it can reward innovations in training algorithms that optimize precision on a fine-granularity during training by boosting performance and energy-efficiency.

Another approach is to modify the training cost function to limit the number of terms per weight value. Processing weights term-serially for $A \times W$ and $G \times W$ training phases would alleviate the synchronization overhead across lanes of the same PE. The incremental network quantization (INQ) [126] is a method that constrains the weights to be either powers of two (one term per value) or zero. However, INQ targets pre-trained networks. Future work can study training networks from scratch with limiting the number of terms per weight value. Further, future work can study the effect of other dataflows, e.g., spatial dataflow where groups of values will be accessed spatially in the WH-dimensions before going depthwise across C-dimension, on the intra-PE synchronization overhead and accordingly performance.

Further, *FPRaker* can benefit and can be directly applied to new quantization techniques such as DRQ [129]. It's presented in the context of inference but can also be applied to training. DRQ quantizes activations per layer on-the-fly through predicting the importance of different regions in the feature maps. It quantizes the unimportant regions to 4b since they use conventional 4b MAC units. *FPRaker*'s ability to adapt to any precision can allow this technique to quantize activations on a finer-grain achieving more benefits.

We have discussed a broad spectrum of software level training acceleration techniques and some accelerator designs in the introduction. Since *FPRaker* is a processing element level method it can in principle be used with any of these techniques to further boost performance and energy efficiency. However, there will be interactions that would need to be investigated. For example, since *FPRaker* requires more parallelism to match and exceed the throughput of a conventional bit-parallel PE it will interact with data reuse/dataflow selection methods. However, data parallelism is abundant during training due to the use of batching. The investigation of these interactions while certainly interesting is left for future work.

Bibliography

[1] NVIDIA Tesla V100 GPU Achitecture. https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf, 2017.

[2] Cerebras CS1. https://www.cerebras.net/product/, 2019.

[3] Gaudi training platform white paper. https://habana.ai/wp-content/uploads/2019/06/Habana-Gaudi-Training-Platform-whitepaper.pdf, 2019.

[4] Chip Design with Deep Reinforcement Learning. https://ai.googleblog.com/2020/04/chip-design-with-deep-reinforcement.html, 2020.

[5] Yaser S. Abu-Mostafa, Malik Magdon-Ismail, and Hsuan-Tien Lin. *Learning From Data*. AMLBook, 2012.

[6] Jorge Albericio, Alberto Delmás, Patrick Judd, Sayeh Sharify, Gerard O'Leary, Roman Genov, and Andreas Moshovos. Bit-pragmatic deep neural network computing. In *Intl' Symp. on Microarchitecture*, 2017.

[7] Jorge Albericio, Patrick Judd, Tayler Hetherington, Tor Aamodt, Natalie Enright Jerger, and Andreas Moshovos. CNVLUTIN: Ineffectual-Neuron-Free Deep Neural Network Computing. In *Intl' Symp. on Computer Architecture*, 2016.

[8] Dan Alistarh, Demjan Grubic, Jerry Li, Ryota Tomioka, and Milan Vojnovic. Qsgd: Communication-efficient sgd via gradient quantization and encoding. In *Advances in Neural Information Processing Systems*, pages 1709–1720, 2017.

[9] Dario Amodei, Danny Hernadez, Girish Sastry, Jack Clark, Greg Brockman, and Ilya Sutskever. Open AI Blog. https://openai.com/blog/ai-and-compute/.

[10] A. D Booth. A signed binary multiplication technique. *The Quarterly Journal of Mechanics and Applied Mathematics*, 4(2):236–240, 1951.

[11] Samuel R. Bowman, Gabor Angeli, Christopher Potts, and Christopher D. Manning. A large annotated corpus for learning natural language inference. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pages 632–642, Lisbon, Portugal, September 2015. Association for Computational Linguistics.

[12] Cadence. Innovus implementation system. https://www.cadence.com/content/cadence-www/global/en_US/home/tools/digital-design-and-signoff/hierarchical-design-and-floorplanning/innovus-implementation-system.html.

[13] Francisco M. Castro, Manuel J. Marín-Jiménez, Nicolás Guil, Cordelia Schmid, and Karteek Alahari. End-to-end incremental learning. In *Computer Vision - ECCV 2018 - 15th European Conference, Munich, Germany, September 8-14, 2018, Proceedings, Part XII*, pages 241–257, 2018.

[14] P Chau, K. Chew, and W. Ki. A bit-serial floating-point complex multiplier-accumulator for fault-tolerant digital signal processing arrays. In *IEEE International Conference on Acoustics, Speech, and Signal Processing*, pages 483–486. IEEE, 1987.

[15] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. In *Intl' Symp. on Computer Architecture*, 2016.

[16] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. In *ACM SIGARCH Computer Architecture News*, volume 44, pages 367–379. IEEE Press, 2016.

[17] Yu-Hsin Chen, Tien-Ju Yang, Joel S. Emer, and Vivienne Sze. Eyeriss v2: A flexible accelerator for emerging deep neural networks on mobile devices. *IEEE J. Emerg. Sel. Topics Circuits Syst.*, 9(2):292–308, 2019.

[18] Kyunghyun Cho, Bart van Merrienboer, Dzmitry Bahdanau, and Yoshua Bengio. On the properties of neural machine translation: Encoder-decoder approaches, 2014.

[19] Jungwook Choi, Zhuo Wang, Swagath Venkataramani, Pierce I-Jen Chuang, Vijayalakshmi Srinivasan, and Kailash Gopalakrishnan. Pact: Parameterized clipping activation for quantized neural networks. *arXiv preprint arXiv:1805.06085*, 2018.

[20] M. Courbariaux, Y. Bengio, and J.-P. David. BinaryConnect: Training Deep Neural Networks with binary weights during propagations. volume abs/1511.00363, 2015.

[21] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Training deep neural networks with low precision multiplications, 2014.

[22] Yann Le Cun, John S. Denker, and Sara A. Solla. Optimal brain damage. In *Advances in Neural Information Processing Systems*, pages 598–605. Morgan Kaufmann, 1990.

[23] Dipankar Das, Naveen Mellempudi, Dheevatsa Mudigere, Dhiraj D. Kalamkar, Sasikanth Avancha, Kunal Banerjee, Srinivas Sridharan, Karthik Vaidyanathan, Bharat Kaul, Evangelos Georganas, Alexander Heinecke, Pradeep Dubey, Jesús Corbal, Nikita Shustrov, Roman Dubtsov, Evarist Fomenko, and Vadim O. Pirogov. Mixed precision training of convolutional neural networks using integer operations. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*, 2018.

[24] Christopher De Sa, Megan Leszczynski, Jian Zhang, Alana Marzoev, Christopher R Aberger, Kunle Olukotun, and Christopher Ré. High-accuracy low-precision training. *arXiv preprint arXiv:1803.03383*, 2018.

[25] Alberto Delmas Lascorz, Patrick Judd, Dylan Malone Stuart, Zissis Poulos, Mostafa Mahmoud, Sayeh Sharify, Milos Nikolic, Kevin Siu, and Andreas Moshovos. Bit-tactical: A software/hardware approach to exploiting value and bit sparsity in neural networks. In *Proceedings of the Twenty-Fourth International*

Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '19, pages 749–763, New York, NY, USA, 2019. ACM.

[26] Tim Dettmers and Luke Zettlemoyer. Sparse networks from scratch: Faster training without losing performance. *arXiv preprint arXiv:1907.04840*, 2019.

[27] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding, 2018.

[28] Mario Drumond, Tao Lin, Martin Jaggi, and Babak Falsafi. Training dnns with hybrid block floating point. In *Proceedings of the 32Nd International Conference on Neural Information Processing Systems*, NIPS'18, pages 451–461, USA, 2018. Curran Associates Inc.

[29] Charles Eckert, Xiaowei Wang, Jingcheng Wang, Arun Subramaniyan, Ravi R. Iyer, Dennis Sylvester, David T. Blaauw, and Reetuparna Das. Neural cache: Bit-serial in-cache acceleration of deep neural networks. In *45th ACM/IEEE Annual Intl' Symp. on Computer Architecture, ISCA 2018, Los Angeles, CA, USA, June 1-6, 2018*, pages 383–396, 2018.

[30] Desmond Elliott, Stella Frank, Khalil Sima'an, and Lucia Specia. Multi30k: Multilingual english-german image descriptions, 2016.

[31] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. Neural architecture search: A survey, 2018.

[32] Ben Feinberg, Uday Kumar Reddy Vengalam, Nathan Whitehair, Shibo Wang, and Engin Ipek. Enabling scientific computing on memristive accelerators. In *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ISCA '18, page 367–382. IEEE Press, 2018.

[33] Sameh Galal and Mark Horowitz. Energy-efficient floating-point unit design. *IEEE Trans. Computers*, 60(7):913–922, 2011.

[34] Mingyu Gao, Jing Pu, Xuan Yang, Mark Horowitz, and Christos Kozyrakis. TETRIS: scalable and efficient neural network acceleration with 3d memory. In *Intl' Conf. on Architectural Support for Programming Languages and Operating Systems*, 2017.

[35] Ross B. Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. *CoRR*, abs/1311.2524, 2013.

[36] Maximilian Golub, Guy Lemieux, and Mieszko Lis. Dropback: Continuous pruning during training. *arXiv preprint arXiv:1806.06949*, 2018.

[37] Ashish Gondimalla, Noah Chesnut, Mithuna Thottethodi, and T. N. Vijaykumar. Sparten: A sparse tensor accelerator for convolutional neural networks. In *Proceedings of the 52Nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, pages 151–165, New York, NY, USA, 2019. ACM.

[38] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, large minibatch sgd: Training imagenet in 1 hour, 2017.

[39] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. Deep learning with limited numerical precision. In *International Conference on Machine Learning*, pages 1737–1746, 2015.

[40] Udit Gupta, Brandon Reagen, Lillian Pentecost, Marco Donato, Thierry Tambe, Alexander M. Rush, Gu-Yeon Wei, and David Brooks. MASR: A modular accelerator for sparse rnns. In *28th International Conference on Parallel Architectures and Compilation Techniques, PACT 2019, Seattle, WA, USA, September 23-26, 2019*, pages 1–14. IEEE, 2019.

[41] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A. Horowitz, and William J. Dally. Eie: Efficient inference engine on compressed deep neural network. In *Intl' Symp. on Computer Architecture*, 2016.

[42] Song Han, Huizi Mao, and William J. Dally. Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding. *CoPR*, abs/1510.00149, 2015.

[43] Awni Y. Hannun, Carl Case, Jared Casper, Bryan C. Catanzaro, Greg Diamos, Erich Elsen, Ryan Prenger, Sanjeev Satheesh, Shubho Sengupta, Adam Coates, and Andrew Y. Ng. Deep speech: Scaling up end-to-end speech recognition. *CoRR*, abs/1412.5567, 2014.

[44] F. Maxwell Harper and Joseph A. Konstan. The movielens datasets: History and context. *ACM Trans. Interact. Intell. Syst.*, 5(4), December 2015.

[45] Sayed Hadi Hashemi, Sangeetha Abdu Jyothi, and Roy H Campbell. Tictac: Accelerating distributed deep learning with communication scheduling. *arXiv preprint arXiv:1803.03288*, 2018.

[46] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015.

[47] Xiangnan He, Lizi Liao, Hanwang Zhang, Liqiang Nie, Xia Hu, and Tat-Seng Chua. Neural collaborative filtering, 2017.

[48] Kartik Hegde, Jiyong Yu, Rohit Agrawal, Mengjia Yan, Michael Pellauer, and Christopher W. Fletcher. Ucnn: Exploiting computational reuse in deep neural networks via weight repetition. In *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ISCA '18, pages 674–687, Piscataway, NJ, USA, 2018. IEEE Press.

[49] Greg Henry, Ping Tak Peter Tang, and Alexander Heinecke. Leveraging the bfloat16 artificial intelligence datatype for higher-precision computations. *2019 IEEE 26th Symposium on Computer Arithmetic (ARITH)*, Jun 2019.

[50] HewlettPackard. Cacti. `https://github.com/HewlettPackard/cacti`.

[51] Geoffrey E. Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan R. Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors, 2012.

[52] Sepp Hochreiter, Yoshua Bengio, Paolo Frasconi, and Jürgen Schmidhuber. Gradient flow in recurrent nets: the difficulty of learning long-term dependencies, 2001.

[53] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. Gpipe: Efficient training of giant neural networks using pipeline parallelism. In *Advances in Neural Information Processing Systems*, pages 103–112, 2019.

[54] Forrest N. Iandola, Matthew W. Moskewicz, Khalid Ashraf, Song Han, William J. Dally, and Kurt Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and <1mb model size. *CoRR*, abs/1602.07360, 2016.

[55] Intel. Modelsim-intel, fpga edition software. `https://www.intel.ca/content/www/ca/en/software/programmableprime/model-sim.html`.

[56] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift, 2015.

[57] Roxana Istrate, Adelmo Cristiano Innocenza Malossi, Costas Bekas, and Dimitrios S. Nikolopoulos. Incremental training of deep convolutional neural networks. *CoRR*, abs/1803.10232, 2018.

[58] Animesh Jain, Amar Phanishayee, Jason Mars, Lingjia Tang, and Gennady Pekhimenko. Gist: Efficient data encoding for deep neural network training. In *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ISCA '18, pages 776–789, Piscataway, NJ, USA, 2018. IEEE Press.

[59] Anand Jayarajan. *Priority-based parameter propagation for distributed deep neural network training*. PhD thesis, University of British Columbia, 2019.

[60] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, pages 1–12, New York, NY, USA, 2017. ACM.

[61] Patrick Judd, Jorge Albericio, Tayler Hetherington, Tor Aamodt, and Andreas Moshovos. Stripes: Bit-serial Deep Neural Network Computing . In *Intl' Symp. on Microarchitecture*, 2016.

[62] Dongyoung Kim, Junwhan Ahn, and Sungjoo Yoo. Zena: Zero-aware neural network accelerator. *IEEE Design Test*, 35:39–46, 2018.

[63] Urs Köster, Tristan J. Webb, Xin Wang, Marcel Nassar, Arjun K. Bansal, William H. Constable, Oğuz H. Elibol, Scott Gray, Stewart Hall, Luke Hornof, Amir Khosrowshahi, Carey Kloss, Ruby J. Pai, and Naveen Rao. Flexpoint: An adaptive numerical format for efficient training of deep neural networks. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, pages 1740–1750, USA, 2017. Curran Associates Inc.

[64] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. *Commun. ACM*, 60(6):84–90, May 2017.

[65] H. T. Kung, Bradley McDanel, and Sai Qian Zhang. Packing sparse convolutional neural networks for efficient systolic array implementations: Column combining under joint optimization. In Iris Bahar, Maurice Herlihy, Emmett Witchel, and Alvin R. Lebeck, editors, *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2019, Providence, RI, USA, April 13-17, 2019*, pages 821–834. ACM, 2019.

[66] Pran Kurup and Taher Abbasi. *Logic Synthesis Using Synopsys*. Springer Publishing Company, Incorporated, 2nd edition, 2011.

[67] Alberto Delmas Lascorz, Sayeh Sharify, Isak Edo Vivancos, Dylan Malone Stuart, Omar Mohamed Awad, Patrick Judd, Mostafa Mahmoud, Milos Nikolic, Kevin Siu, Zissis Poulos, and Andreas Moshovos. Shapeshifter: Enabling fine-grain data width adaptation in deep learning. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2019, Columbus, OH, USA, October 12-16, 2019*, pages 28–41. ACM, 2019.

[68] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, 05 2015.

[69] Jinmook Lee, Changhyeon Kim, Sanghoon Kang, Dongjoo Shin, Sangyeob Kim, and Hoi-Jun Yoo. UNPU: an energy-efficient deep neural network accelerator with fully variable weight bit precision. *J. Solid-State Circuits*, 54(1):173–185, 2019.

[70] H. Liao, J. Tu, J. Xia, and X. Zhou. DaVinci: A scalable architecture for neural network computing. In *2019 IEEE Hot Chips 31 Symposium (HCS)*, pages 1–44, 2019.

[71] Aristidis Likas, Nikos Vlassis, and Jakob Verbeek. The global k-means clustering algorithm. *Pattern Recognition*, 36:451–461, 08 2002.

[72] Tsung-Yi Lin, Michael Maire, Serge Belongie, Lubomir Bourdev, Ross Girshick, James Hays, Pietro Perona, Deva Ramanan, C. Lawrence Zitnick, and Piotr Dollár. Microsoft coco: Common objects in context, 2014.

[73] Yujun Lin, Song Han, Huizi Mao, Yu Wang, and William Dally. Deep gradient compression: Reducing the communication bandwidth for distributed training. 2018.

[74] Chang Liu, Changhu Wang, Fuchun Sun, and Yong Rui. Image2text: A multimodal image captioner. In *Proceedings of the 24th ACM International Conference on Multimedia*, MM '16, page 746–748, New York, NY, USA, 2016. Association for Computing Machinery.

[75] Shaoli Liu, Zidong Du, Jinhua Tao, Dong Han, Tao Luo, Yuan Xie, Yunji Chen, and Tianshi Chen. Cambricon: An instruction set architecture for neural networks. In *2016 IEEE/ACM Intl' Conf. on Computer Architecture (ISCA)*, 2016.

[76] Sangkug Lym, Armand Behroozi, Wei Wen, Ge Li, Yongkee Kwon, and Mattan Erez. Mini-batch serialization: Cnn training with inter-layer data reuse, 2018.

[77] Pavan Kumar Mallapragada, Rong Jin, and Anil Jain. Non-parametric mixture models for clustering. In Edwin R. Hancock, Richard C. Wilson, Terry Windeatt, Ilkay Ulusoy, and Francisco Escolano, editors, *Structural, Syntactic, and Statistical Pattern Recognition*, pages 334–343, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[78] Peter Mattson, Christine Cheng, Cody Coleman, Greg Diamos, Paulius Micikevicius, David Patterson, Hanlin Tang, Gu-Yeon Wei, Peter Bailis, Victor Bittorf, David Brooks, Dehao Chen, Debojyoti Dutta, Udit Gupta, Kim Hazelwood, Andrew Hock, Xinyuan Huang, Atsushi Ike, Bill Jia, Daniel Kang, David Kanter, Naveen Kumar, Jeffery Liao, Guokai Ma, Deepak Narayanan, Tayo Oguntebi, Gennady Pekhimenko, Lillian Pentecost, Vijay Janapa Reddi, Taylor Robie, Tom St. John, Tsuguchika Tabaru, Carole-Jean Wu, Lingjie Xu, Masafumi Yamazaki, Cliff Young, and Matei Zaharia. Mlperf training benchmark, 2019.

[79] Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. Pointer sentinel mixture models. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*, 2017.

[80] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory F. Diamos, Erich Elsen, David García, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, and Hao Wu. Mixed precision training. In *6th International Conference on Learning Representations, 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*, 2018.

[81] Inc. Micron Technology. Ddr4 power calculator 4.0. `https://www.micron.com/~/media/documents/products/power-calculator/ddr4_power_calc.xlsm`.

[82] Hesham Mostafa and Xin Wang. Parameter efficient training of deep convolutional neural networks by dynamic sparse reparameterization. In *International Conference on Machine Learning*, pages 4646–4655, 2019.

[83] Hesham Mostafa and Xin Wang. Parameter efficient training of deep convolutional neural networks by dynamic sparse reparameterization. In *International Conference on Machine Learning*, pages 4646–4655, 2019.

[84] Manish Munikar, Sushil Shakya, and Aakash Shrestha. Fine-grained sentiment classification using bert, 2019.

[85] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, Phillip B. Gibbons, and Matei Zaharia. Pipedream: Generalized pipeline parallelism for dnn training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, pages 1–15, New York, NY, USA, 2019. ACM.

[86] NVIDIA. Training with mixed precision. `https://docs.nvidia.com/deeplearning/sdk/mixed-precision-training/index.html`.

[87] Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Brucek Khailany, Joel Emer, Stephen W. Keckler, and William J. Dally. SCNN: an accelerator for compressed-sparse convolutional neural networks. In *Intl' Symp. on Computer Architecture*, ISCA '17, 2017.

[88] Eunhyeok Park, Dongyoung Kim, and Sungjoo Yoo. Energy-efficient neural network accelerator based on outlier-aware low-precision computation. In *Intl' Symp. on Computer Architecture*, 2018.

[89] Eunhyeok Park, Sungjoo Yoo, and Peter Vajda. Value-aware quantization for training and inference of neural networks, 2018.

[90] Gennady Pekhimenko, Vivek Seshadri, Onur Mutlu, Phillip B. Gibbons, Michael A. Kozuch, and Todd C. Mowry. Base-delta-immediate compression: Practical data compression for on-chip caches. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, page 377–388, New York, NY, USA, 2012. Association for Computing Machinery.

[91] Yanghua Peng, Yibo Zhu, Yangrui Chen, Yixin Bao, Bairen Yi, Chang Lan, Chuan Wu, and Chuanxiong Guo. A generic communication scheduler for distributed dnn training acceleration. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, pages 16–29, New York, NY, USA, 2019. ACM.

[92] Intel AI PlaidML. Plaidml. 2017.

[93] Tomaso Poggio, Hrushikesh Mhaskar, Lorenzo Rosasco, Brando Miranda, and Qianli Liao. Why and when can deep – but not shallow – networks avoid the curse of dimensionality: a review, 2016.

[94] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. Squad: 100,000+ questions for machine comprehension of text, 2016.

[95] Minsoo Rhu, Mike O'Connor, Niladrish Chatterjee, Jeff Pool, Youngeun Kwon, and Stephen W Keckler. Compressing dma engine: Leveraging activation sparsity for training deep neural networks. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 78–91. IEEE, 2018.

[96] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. *Learning Internal Representations by Error Propagation*, page 318–362. MIT Press, Cambridge, MA, USA, 1986.

[97] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *CoRR*, abs/1409.0575, September 2014.

[98] Charbel Sakr, Naigang Wang, Chia-Yu Chen, Jungwook Choi, Ankur Agrawal, Naresh Shanbhag, and Kailash Gopalakrishnan. Accumulation bit-width scaling for ultra-low precision training of deep networks, 2019.

[99] Charbel Sakr, Naigang Wang, Chia-Yu Chen, Jungwook Choi, Ankur Agrawal, Naresh Shanbhag, and Kailash Gopalakrishnan. Accumulation bit-width scaling for ultra-low precision training of deep networks, 2019.

[100] Frank Seide, Hao Fu, Jasha Droppo, Gang Li, and Dong Yu. 1-bit stochastic gradient descent and application to data-parallel distributed training of speech dnns. In *Interspeech 2014*, September 2014.

[101] Christopher J Shallue, Jaehoon Lee, Joseph Antognini, Jascha Sohl-Dickstein, Roy Frostig, and George E Dahl. Measuring the effects of data parallelism on neural network training. *Journal of Machine Learning Research*, 20(112):1–49, 2019.

[102] Sayeh Sharify, Alberto Delmas Lascorz, Mostafa Mahmoud, Milos Nikolic, Kevin Siu, Dylan Malone Stuart, Zissis Poulos, and Andreas Moshovos. Laconic deep learning inference acceleration. In *Proceedings of the 46th International Symposium on Computer Architecture, ISCA 2019, Phoenix, AZ, USA, June 22-26, 2019*, pages 304–317, 2019.

[103] Sayeh Sharify, Alberto Delmas Lascorz, Kevin Siu, Patrick Judd, and Andreas Moshovos. Loom: Exploiting weight and activation precisions to accelerate convolutional neural networks. In *Proceedings of the 55th Annual Design Automation Conference*, page 20. ACM, 2018.

[104] Hardik Sharma, Jongse Park, Naveen Suda, Liangzhen Lai, Benson Chau, Vikas Chandra, and Hadi Esmaeilzadeh. Bit fusion: Bit-level dynamically composable architecture for accelerating deep neural network. In *ISCA*, pages 764–775. IEEE Computer Society, 2018.

[105] Jitesh R. Shinde and Suresh S. Salankar. VLSI implementation of bit serial architecture based multiplier in floating point arithmetic. In *2015 International Conference on Advances in Computing, Communications and Informatics, ICACCI 2015, Kochi, India, August 10-13, 2015*, pages 1672–1677. IEEE, 2015.

[106] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using gpu model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

[107] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[108] Jasper Snoek, Hugo Larochelle, and Ryan P. Adams. Practical bayesian optimization of machine learning algorithms. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'12, page 2951–2959, Red Hook, NY, USA, 2012. Curran Associates Inc.

[109] Linghao Song, Jiachen Mao, Youwei Zhuo, Xuehai Qian, Hai Li, and Yiran Chen. Hypar: Towards hybrid parallelism for deep learning accelerator array. *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2019.

[110] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *J. Mach. Learn. Res.*, 15(1):1929–1958, January 2014.

[111] Xu Sun, Xuancheng Ren, Shuming Ma, and Houfeng Wang. meprop: Sparsified back propagation for accelerated deep learning with reduced overfitting. In *Proceedings of the 34th International Conference on Machine Learning - Volume 70*, ICML'17, pages 3299–3308. JMLR.org, 2017.

[112] Richard S. Sutton and Andrew G. Barto. *Introduction to Reinforcement Learning*. MIT Press, Cambridge, MA, USA, 1st edition, 1998.

[113] Ehsan Taher, Seyed Abbas Hoseini, and Mehrnoush Shamsfard. Beheshti-ner: Persian named entity recognition using bert, 2020.

[114] Igor V. Tetko, David J. Livingstone, and Alexander I. Luik. Neural network studies, 1. comparison of overfitting and overtraining. *Journal of Chemical Information and Computer Sciences*, 35:826–833, 1995.

[115] Swagath Venkataramani, Ashish Ranjan, Subarno Banerjee, Dipankar Das, Sasikanth Avancha, Ashok Jagannathan, Ajaya Durg, Dheemanth Nagaraj, Bharat Kaul, Pradeep Dubey, and Anand Raghunathan. Scaledeep: A scalable compute architecture for learning and evaluating deep networks. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, pages 13–26, New York, NY, USA, 2017. ACM.

[116] Oriol Vinyals, Alexander Toshev, Samy Bengio, and Dumitru Erhan. Show and tell: Lessons learned from the 2015 MSCOCO image captioning challenge. *CoRR*, abs/1609.06647, 2016.

[117] Naigang Wang, Jungwook Choi, Daniel Brand, Chia-Yu Chen, and Kailash Gopalakrishnan. Training deep neural networks with 8-bit floating point numbers. In *Proceedings of the 32Nd International Conference on Neural Information Processing Systems*, NIPS'18, pages 7686–7695, USA, 2018. Curran Associates Inc.

[118] Shibo Wang and Pankaj Kanwar. Bfloat16: The secret to high performance on cloud tpus, 2019.

[119] Zelun Wang and Jyh-Charn Liu. Translating math formula images to latex sequences using deep neural networks with sequence-level training, 2019.

[120] Wei Wen, Cong Xu, Feng Yan, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. Terngrad: Ternary gradients to reduce communication in distributed deep learning. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, pages 1508–1518, USA, 2017. Curran Associates Inc.

[121] Yuxin Wu, Alexander Kirillov, Francisco Massa, Wan-Yen Lo, and Ross Girshick. Detectron2. `https://github.com/facebookresearch/detectron2`, 2019.

[122] Tianjun Xiao, Jiaxing Zhang, Kuiyuan Yang, Yuxin Peng, and Zheng Zhang. Error-driven incremental learning in deep convolutional neural network for large-scale image classification. In *Proceedings of the ACM International Conference on Multimedia, MM '14, Orlando, FL, USA, November 03 - 07, 2014*, pages 177–186, 2014.

[123] Jiaqi Zhang, Xiangru Chen, Mingcong Song, and Tao Li. Eager pruning: Algorithm and architecture support for fast training of deep neural networks. In *Proceedings of the 46th International Symposium on Computer Architecture*, ISCA '19, pages 292–303, New York, NY, USA, 2019. ACM.

[124] Shijin Zhang, Zidong Du, Lei Zhang, Huiying Lan, Shaoli Liu, Ling Li, Qi Guo, Tianshi Chen, and Yunji Chen. Cambricon-x: An accelerator for sparse neural networks. In *Intl' Symp. on Microarchitecture*, 2016.

[125] Zhiyuan Zhang, Pengcheng Yang, Xuancheng Ren, and Xu Sun. Memorized sparse backpropagation, 2019.

[126] Aojun Zhou, Anbang Yao, Yiwen Guo, Lin Xu, and Yurong Chen. Incremental network quantization: Towards lossless cnns with low-precision weights, 2017.

[127] Shuchang Zhou, Yuxin Wu, Zekun Ni, Xinyu Zhou, He Wen, and Yuheng Zou. Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients. *arXiv preprint arXiv:1606.06160*, 2016.

[128] Xuda Zhou, Zidong Du, Qi Guo, Chengsi Liu, Chao Wang, Xuehai Zhou, Ling Li, Tianshi Chen, and Yunji Chen. Cambricon-S: addressing irregularity in sparse neural networks through a cooperative software/hardware approach. In *Intl' Symp. on Microarchitecture*, 2018.

[129] Feiyang Wu Zhaoming Jiang Li Jiang Naifeng Jing Xiaoyao Liang Zhuoran Song, Bangqi Fu. Drq: Dynamic region-based quantization for deep neural network acceleration. In *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ISCA '20, 2020.