Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Quantized Backpropagation for Sparse Transfer Learning

Master Thesis

Berke Egeli

September 18, 2023

Advisors: Prof. Dr. M. Püschel, T. Pegolotti
Department of Computer Science, ETH Zürich

Abstract

Neural networks are growing at exponential rates. It is becoming more resource intensive and time consuming to train and deploy more sophisticated models. In order to alleviate this issue, many model compression methods have been developed. These methods make model development both faster and cheaper. Two such methods are quantization and pruning. Unfortunately, most quantization and pruning methods remain theoretical without system support. This thesis introduces a high-performance library with quantized and sparse kernels to speed up neural network finetuning. Our library provides efficient versions of the backpropagation algorithm for linear and convolutional modules. Our algorithms apply to unstructured sparsity and implement 8-bit integer quantization kernels for forward and backward passes. Models trained using our framework provide up to 2x-4x speed ups while halving the memory allocation with acceptable accuracy loss.

Contents

Contents ii

List of Figures iv

List of Tables v

1 Introduction 1
1.1 Contributions . . . 1
1.2 Related Work . . . 2

2 Background 3
2.1 Backpropagation and Transfer Learning . . . 3
2.1.1 Forward Pass . . . 3
2.1.2 Backward Pass . . . 4
2.1.3 Transfer Learning . . . 4
2.2 Quantization . . . 5
2.3 Pruning . . . 9

3 Quantization and Sparse Kernel Implementations 13
3.1 Quantization Kernels . . . 13
3.1.1 Quantized Representation . . . 13
3.1.2 SAWB Quantization . . . 15
3.1.3 Dithered Quantization . . . 17
3.1.4 Quantization Granularity . . . 21
3.2 Sparse Propagation Kernels . . . 21
3.2.1 Sparse Data Model for Linear Layers . . . 22
3.2.2 Sparse Data Model for Convolutional Layers . . . 22
3.2.3 Linear Forward Propagation . . . 22
3.2.4 Linear Backward Propagation . . . 23
3.2.5 Convolutional Forward Propagation . . . 25
3.2.6 Convolutional Backward Propagation . . . 26
3.3 Multi-threaded Implementation . . . 27
3.3.1 Multi-threaded Quantization Kernels . . . 28
3.3.2 Multi-threaded Propagation Kernels . . . 29

4 Evaluation 31
4.1 Accuracy . . . 32
4.2 Timing Performance . . . 33
4.2.1 Module Timing Performance . . . 33

4.2.2 End-to-end Timing Performance 37
4.3 Memory Usage 39
4.3.1 Memory Allocation 39
4.3.2 Data Transfers 42

5 Conclusion **44**

A Stochastic/Deterministic Quantization **45**

B Packing/Unpacking Algorithms **47**

Bibliography **50**

List of Figures

2.1 Visualization of a Quantization Function 6
2.2 Visual representations of COO, CSF, and F-COO formats 11
2.3 Visual representation of HiCOO format for a three-dimensional sparse tensor . 12

3.1 Packing algorithm visualization 14
3.2 Unpacking algorithm visualization 15
3.3 CSR representation of a sparse matrix 22
3.4 CSF representation of a sparse tensor 23

4.1 Validation Accuracy of QSparseProp 32
4.2 Runtimes of single-threaded dense, sparse and quantized, sparse linear modules for forward and backward passes on various sparsities 33
4.3 Runtimes of multi-threaded dense, sparse and quantized, sparse linear modules for forward and backward passes on various sparsities 34
4.4 Runtimes of single-threaded forward and backward propagations of dense, sparse and quantized, sparse convolutional modules 35
4.5 Runtimes of multi-threaded forward and backward propagations of dense, sparse and quantized, sparse convolutional modules 36
4.6 Finetuning speedup comparison of sparse and quantized, sparse ResNet50 with respect to dense PyTorch model 38
4.7 Memory allocations of linear and convolutional modules during the forward and backward passes 40
4.8 Memory allocated during the forward and backward passes of a ResNet50 model 41
4.9 Total simulated data transfers from memory to L3 cache during the forward and backward passes of a ResNet50 model 42

List of Tables

3.1 Symbols used to denote the dimensions of tensors in convolutional layers. . . . 25

4.1 Speedups for sparse and quantized, sparse convolutional modules with respect to dense PyTorch modules with input tensor dimensions (B, IC, M, N) = (64, 128, 8, 8) 37

4.2 Speedups for sparse and quantized, sparse convolutional modules with respect to dense PyTorch modules with input tensor dimensions (B, IC, M, N) = (64, 128, 122, 122) 37

Chapter 1

Introduction

Over the past decade, machine learning models started to solve increasingly more complex, human-like tasks, making them a very hot research topic. However, this increased utility usually came with an increased model size as well [24].

An early convolutional neural network, LeNet-5, used to classify handwritten digits had around 1M parameters [36]. Fast-forward to 2012, the model that won the ImageNet competition had around 60M parameters [34]. More recently, large language model GPT-3 released in 2020 had 175B parameters [4]. In only three years, its successor GPT-4 was already around an order of magnitude bigger with 1.76T parameters [49]. This exponential growth in model sizes makes model training, deployment, and inference a very resource-intensive and time-consuming endeavor. Therefore, there has been a renewed interest in model compression methods such as quantization and pruning to accelerate model training and inference, and to reduce the resource requirements to make it accessible to a wider range of hardware. However, most of the quantization and pruning methods introduced so far remain theoretical with little system support.

Quantization methods introduced are usually either simulations which do not actually improve performance [7, 65] or are intended for inference only [21] which ignores a large part of the model life cycle, the training. Similarly, pruning methods usually lack system support as well, as most of the pruning methods induce unstructured sparsity which is difficult to use to improve performance [47].

1.1 Contributions

This thesis introduces a high-performance parallel, SIMD library for neural network fine-tuning on commodity CPUs. Our contributions are:

1. SIMD implementations of SAWB [8, 9] and Dithered [65] quantization methods.
2. 8-bit integer quantization implementation of Dithered quantization which is a floating point quantization scheme.
3. Extension of the sparse training library SparseProp [47] with quantization support.
4. Parallel, SIMD linear and convolution kernels for quantized, sparse forward and backward propagations on commodity CPU.

We focus on image classification tasks to test our library. We achieve around 3x and 4x speed up in forward and backward propagations with respect to dense PyTorch models using 90% sparse, 8-bit quantized models. Compared to SparseProp, our quantized, sparse models achieve around 1.1x and 1.2x speed up during forward and backward propagations

at 90% sparsity. Our library maintains on average 96.5% of the dense model accuracy for the tested datasets.

We focused on finetuning tasks on commodity CPUs as it highlights the benefits of model compression methods, namely training neural networks on resource constraint environments. Even though training and inference are still faster on GPUs, it is not always reasonable to expect users to have access to specialized hardware like that. Moreover, pruning and quantization methods improved such that models trained using such methods can provide competitive speeds on CPU against models running on GPU [15]. Similarly, while training from scratch can lead to better accuracy results, it is not always feasible for various reasons such as the cost of training a model or difficulty of collecting or accessing data. Therefore, finetuning using pre-trained models highlights an important use case for quantization and pruning.

1.2 Related Work

Quantization There have been many quantization schemes developed over the years. As it is not possible to go over every development in quantization, we refer the interested users to [41, 21]. In this thesis, we implement two quantization methods, SAWB [9] and Dithered quantization [65].

SAWB is a symmetric integer quantization function. It quantizes tensors based on the best fit of various probability distributions to predict the distribution of intermediate outputs which reduce quantization noise. We explain the details of this quantization method and how we implement it using SIMD instructions in subsection 3.1.2.

Dithered quantization is a floating point quantization method. It uses the standard deviation of the input tensors to quantize them. Depending on how you choose quantization related hyperparameters, it induces unstructured sparsity in the quantized tensors. Owing to the floating point quantization, it does not have a way of controlling the number of bits used in quantized values like most integer quantization methods do. We discuss dithered quantization in more depth in subsection 3.1.3.

ZipML [69] quantization framework used to quantize activations, weights, and gradients during neural network training. It uses a double sampling technique to unbias the quantized tensors which is essential to successful gradient quantization. It only works for linear models.

Clover [58] is a quantized linear algebra framework that offers a variety of quantization options ranging from 16-bit quantization to 4-bit quantization to perform popular linear algebra operations such as dot product and matrix-vector multiplications quickly. It uses a rather small, fixed quantization granularity to minimize the quantization error.

SparseProp SparseProp [47] is a high-performance parallel, SIMD library for sparse neural network training. It supports both training from scratch and finetuning. It implements high-performance kernels for linear and convolutional layers of neural networks. It uses a compressed sparse fiber (CSF) format, which is an extension of the compressed sparse row format for high dimensional tensors, to represent the unstructured sparsity of the weights.

HiCOO Hierarchical Coordinate (HiCOO) [39] is a sparse storage format for high dimensional tensors with unstructured sparsity. It combines CSF and COO formats to create a mode-agnostic sparse format. This allows it to perform sparse computations by arranging and accessing dimensions in any orientation without penalty. We talk more about HiCOO in 2.3.

Chapter 2

Background

In this chapter, we introduce the theoretical foundations that are essential to understanding our library described in Chapter 3. We start by describing definitions and concepts related to backpropagation algorithm and transfer learning. As the paper's main focus is not machine learning or neural networks, we do not go into much detail. Then, we introduce concepts of quantization. Next, we look into pruning in neural networks and sparse tensor representations.

2.1 Backpropagation and Transfer Learning

Training of neural networks is done through an algorithm called backpropagation. The backpropagation algorithm consists of two steps, forward and backward steps. For the descriptions of forward and backward propagation we use the definitions in [1] and the notations from [30].

2.1.1 Forward Pass

Forward propagation, also known as inference, passes a batch of inputs through the neural network. The final output will be computed through a cascade of computations across the network layers using the newest weights [1]. The network can be optimized for accuracy by comparing this output against the expected output of the training data by using a loss function. A loss function can be defined as a function that maps a value or an event to a real number which can be used to penalize the said event for the incorrect predictions of the model [25]. The output of the loss function will be later used in the backward pass to optimize the model.

Now, we define the forward pass more formally. Imagine a neural network with L layers. We define the input layer (denoted with superscript 0) as

$$\mathbf{v}^{(0)} = [\mathbf{x}, 1], \tag{2.1}$$

where $\mathbf{x}$ is the input data and *1* represents the bias node. The parameters of each layer l are denoted by $\mathbf{W}^{(l)}$. We define an activation function φ which introduces non-linearity into the network to capture and learn complex relationships. The activation function can be various functions such as identity, sigmoid, tanh, rectified linear unit (ReLU), etc. The pre-activation values of layer l are denoted by $\mathbf{z}^{(l)}$. Then, the activated values of layer l are denoted as $\mathbf{v}^{(l)}$. Then for each layer $l = 1 : L - 1$ we can define the pre-activation and activation values as

$$\mathbf{z}^l = f(\mathbf{W}^{(l)}, \mathbf{v}^{(l-1)}) \; and \; \mathbf{v}^{(l)} = [\varphi(\mathbf{z}^l), 1], \tag{2.2}$$

where *f(.,.)* represents the matrix-matrix multiplication for fully connected layers and convolution operation for convolutional layers. The final layer is usually a fully connected layer, so we have for layer L we have

$$\mathbf{y} = \mathbf{W}^{(L)}\mathbf{v}^{(L-1)} \tag{2.3}$$

$\mathbf{y}$ can be used for regression and classification tasks for inference.

2.1.2 Backward Pass

As we mentioned, in the backward pass we compute the gradients of a loss function ℓ with respect to weights to optimize the parameters. The phase is called the backward pass as we compute the gradients starting from the last layer until we reach the first layer [54]. Computing the gradients this way offers computational benefits as we can use the next layers' gradients to compute the current layer's gradient. Backward pass can be considered dynamic programming as we compute the gradients of upper layers to compute the current layer's gradient.

Now we describe backward pass more formally. Assume we have neural network NN($\mathbf{x}, \mathbf{W}$) where $\mathbf{W} = \{\mathbf{W}^{(i)}\}_{i=1}^{L}$. Then for the loss $\ell(\mathbf{x}, \mathbf{y}, \mathbf{W})$ we compute the gradients with respect to $\mathbf{W}^{(i)}$.

For the final layer L we have

$$\begin{aligned} &\text{Error term: } \delta^{(L)} = \nabla_{NN}\ell, \\ &\text{Gradient: } \nabla_{W^{(L)}}\ell = \delta^{(L)}\mathbf{v}^{(L-1)T}. \end{aligned} \tag{2.4}$$

For hidden layers $l = L - 1 : -1 : 1$ we have

$$\begin{aligned} &\text{Error term: } \delta^{(l)} = \varphi'(\mathbf{z}^{(l)}) \odot (\mathbf{W}^{(l+1)T}\delta^{(l+1)}), \\ &\text{Gradient: } \nabla_{W^{(l)}}\ell = \delta^{(l)}\mathbf{v}^{(l-1)T}. \end{aligned} \tag{2.5}$$

Later, the gradients $\nabla_{W^{(l)}}\ell$ are applied to weights using optimization methods, usually gradient descent methods such as Adaptive Moment Estimation (Adam) [29] or Stochastic Gradient Descent (SGD) [53], to reduce loss ℓ and increase accuracy.

2.1.3 Transfer Learning

As the primary objective of the thesis is developing quantized sparse kernels for transfer learning, we find it imperative to at least mention what transfer learning is. This subsection introduces transfer learning, describes why it is needed and formally defines it. As the primary focus of the thesis isn't on transfer learning, however, we will not discuss it in depth. Interested readers can refer to [50, 64]. The definitions and notations in this subsection are taken from [50, 64].

Transfer learning is a method to train new models using a pre-trained model trained on a similar domain. As neural networks get larger and more complex, working on more complex tasks, they require more data. Unfortunately, data is not always readily available. This might be because data is expensive, rare, or difficult to collect [64]. Therefore, there is

a lot of merit in reusing already trained models that might be trained on a different target dataset but similar in the type of problems it aims to solve.

To formally define transfer learning, we denote the notations introduced in [50]. Denote a domain as $\mathcal{D} = \{\mathcal{X}, P(X)\}$ where $\mathcal{X}$ is a feature set and P(X) is a marginal probability distribution with $X = \{x_1, x_2, ..., x_n\} \in \mathcal{X}$. n is the number of feature vectors in X and $\mathcal{X}$ is the set of all possible feature vectors. Next, we define a task $\mathcal{T}$ for a given domain $\mathcal{D}$. A task is composed of two parts, a label space $\mathcal{Y}$ and a predictive function f(.) learned from pairs of feature vectors and labels $\{x_i, y_i\}$ where $x_i \in \mathcal{X}$ and $y_i \in \mathcal{Y}$. So, a task is defined as $\mathcal{T} = \{\mathcal{Y}, f(.)\}$. Now, we define source and target domain, $\mathcal{D}_s$ and $\mathcal{D}_t$.

$$\begin{aligned} \mathcal{D}_s &= \{(x_{S1}, y_{S1}), (x_{S2}, y_{S2}), ..., (x_{Sn}, y_{Sn})\}, \\ \mathcal{D}_t &= \{(x_{T1}, y_{T1}), (x_{T2}, y_{T2}), ..., (x_{Tn}, y_{Tn})\} \end{aligned} \tag{2.6}$$

Here, x_{Si} and x_{Ti} are the ith data instances of the source and target domains. Similarly, y_{Si} and y_{Ti} are their corresponding labels. We denote the source and target tasks as $\mathcal{T}_s$ and $\mathcal{T}_t$ respectively. The source and target predictive functions are denoted as $f_s(.)$ and $f_t(.)$ respectively.

We now formally define *transfer learning*. For a source domain $\mathcal{D}_s$ and its task $\mathcal{T}_s$, a target domain $\mathcal{D}_t$ and its task $\mathcal{T}_t$, transfer learning hopes to improve the target predictive function $f_t(.)$ in $\mathcal{D}_s$ by using the related information in $\mathcal{D}_s$ and $\mathcal{T}_s$ where $\mathcal{D}_s \neq \mathcal{D}_t$ and $\mathcal{T}_s \neq \mathcal{T}_t$ [50].

2.2 Quantization

Quantization is defined as the process of approximating a continuous signal into a set of discrete symbols or integers [41]. Historically, it has been used in mathematics and signal processing to map large continuous domains to smaller, often finite, discrete ones [21] for approximations and digitalizing continuous variables. More recently, however, quantization has become a topic of machine learning research for neural network inference and training because of its computational implications. For our purposes, we can view quantization as reducing the bit precision of floating point numbers to decrease power and bandwidth usage to improve performance. As the topic of quantization is a vast one, we will only touch on concepts relevant to our work. Interested readers can refer to [21] and [41] for an in-depth survey of quantization methods. Unless stated otherwise, we use the definitions from [21, 41] in this section.

For neural network training, generally, 32-bit floating point numbers (FP32) have been used for most of its history [41]. However, 32 bits usually have more precision than needed for many networks. Starting in the 2010s, quantization has become a more and more important part of machine learning research when it was shown that 8-bit integer quantization (INT8) can be used to speed up inference without degrading accuracy [60]. With this in mind, we can now more formally introduce the problem statement quantization tries to solve.

Problem Statement For simplicity, in our problem statement we will focus on supervised learning, however, the problem statement can easily be extended to other types of learning tasks. For a neural network NN with L layers, assume it has a set of learnable parameters $\theta = \{\theta_1, \theta_2, ..., \theta_L\}$. The objective of supervised learning is to minimize an empirical risk minimization function of the following form:

$$L(\theta) = \frac{1}{N} \sum_{i}^{N} l(x_i, y_i, \theta). \tag{2.7}$$

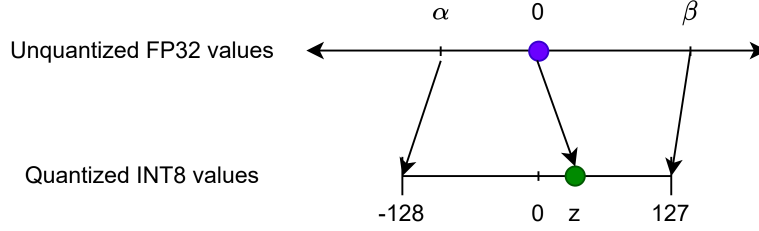


Figure 2.1: Visualization of the quantization function in equation 2.8. (α, β) represents the range of values that can be quantized without clipping. Z represents the value the zero point will take after quantization. The figure is modeled after Figure 3.1 in [63].

Here, we have N samples $\{x_1, x_2..., x_N\}$ and their corresponding labels $\{y_1,y_2, ...,y_N\}$. $l(.,.,.)$ is a loss function such as Mean Squared Error (MSE) or Cross Entropy loss. For the ith layer denote h_i as the input hidden activations, a_i as the output hidden activations, gh_i as the input gradients, and ga_i as the output gradients. If a network is trained with FP32 precision, then naturally its parameters θ are stored in single precision floating point numbers. The objective of the quantization process is to reduce the precision of θ, h, a, gh, and ga to reduce the power and bandwidth consumption to increase inference and training performance without decreasing the model accuracy. The precision reduction is done through quantization functions.

Quantization Function A quantization function $Q : \mathbb{R} \rightarrow \mathbb{Z}$ is a function that takes higher bit precision floating point numbers and returns an output with lower bit precision. In practice, the lower-bit outputs are represented with integer variables, i.e. INT8, due to a lack of hardware support and standardization for low-bit precision floating point numbers. However, in theory, there are a variety of lower-bit floating representations [45, 48], i.e. 8-bit floating point numbers (FP8), and corresponding quantization functions [7, 65].

INT8 and similar integer quantization functions usually have the following form:

$$Q(r) = \text{Int}(r/S) - Z \tag{2.8}$$

Here, S is a floating point scale and Z is an integer zero point. $Int(.)$ function converts its floating point arguments into integer representations. It can be a variety of functions like floor, ceiling, round-to-nearest, or truncate. It is important to notice that, in order to enforce a certain bit precision, it might be necessary to clip the quantization output. For example, to ensure r is quantized into a signed integer using an INT8 quantization function with $Z = 0$ we have to ensure the quantized value is in the range [-128,127]. We visualize the quantization function in Figure 2.1.

This type of quantization function is also referred to as a uniform quantization function as the quantization levels are spread uniformly within the quantized value range. However, there are alternative quantization functions with more dynamic ranges to cover larger value ranges to better represent the inputs like [7]. However, as non-uniform quantization functions are out of the scope of this thesis, we will not discuss them any further.

Dequantization Function Dequantization function is the inverse operation of the quantization function. It is used to recover an approximate floating point representation from the quantized representation. We can denote the dequantization function in the following way:

$$\tilde{r} = DQ(Q(r) + Z) \tag{2.9}$$

It is important to notice that a floating point number r is not equal to $DQ(Q(r))$ because of the rounding errors. This discrepancy is a major source of noise in neural network training and therefore, has implications for model accuracy. We call this the quantization error or noise.

Symmetric/Asymmetric Quantization Quantization functions where the zero point Z is selected to be equal to 0 are called symmetric quantization functions. Conversely, quantization functions where Z is not 0 are called asymmetric. Symmetric quantization functions can be preferable over asymmetric quantization functions due to their simplicity and reduced computational cost which is why in section 3.1 we will use symmetric quantization functions to implement quantization in our library. Asymmetric quantization functions are preferable when the range of values a floating point number r can take could be asymmetric and skewed.

Choosing Quantization Scaling Factor Choosing an appropriate way to compute the quantization scale is perhaps one of the most essential aspects of designing quantization functions. Quantization scales effectively divide a finite range of values into partitions so that floating point numbers can be mapped to one of those partitions. A common way of choosing the scaling factor for integer quantization is the following:

$$S = \frac{\beta - \alpha}{2^b - 1} \tag{2.10}$$

Here, $[\alpha, \beta]$ represents the clipping range which is used to clip real values r. b denotes the bit width of the quantized value. Choosing α and β has important effects on the accuracy of the model and quantization error. A simple way to choose them is to set α to r_{min} and set β to r_{max} where r_{min} and r_{max} are the minimum and maximum values the quantization inputs can take. This approach works well for asymmetric quantization. For symmetric quantization the alternative is setting $-\alpha = \beta = max(|r_{min}|, |r_{max}|)$. Unfortunately, neither of these approaches are without their problems. Imagine a scenario where there are outliers on either extreme. Then setting the clipping bounds to the maximum or minimum will lead to a skew in the quantized values, pushing together real values that are far away from the extremes. To address this issue, α and β can be chosen using percentiles, i.e. choose x^{th} percentile for α, choose $100 - x^{th}$ percentile for β etc. Alternatively, using quality measures like KL divergence and MSE between real values and quantized values is also possible.

Quantization Granularity An important aspect to determine for quantization is to decide the quantization granularity. As we mentioned previously, scaling factors of quantization functions partition finite ranges so that real values can be mapped into these partitions. As the scaling factor is usually determined by the properties of the input values, skews, and outliers can have significant effects on the error. Imagine an input tensor where most of the values are grouped around a real value r and you have one outlier that is much larger or smaller than r. Then, the scaling factor computed from this outlier will map most of the values to the same partitions. During dequantization, values mapped into the same partitions will be assigned the same values, possibly leading to a larger quantization noise than it would have had if the outlier was not present. In addition to the heuristics mentioned above, using different granularities for quantization functions can help us choose better scaling factors. We can group the input values in different granularities and quantize them separately to compute more representative scaling factors that induce lower quantization error. In most cases, the quantization error should be no worse than using a single group.

There are various approaches to choosing quantization granularity. For convolutional neural networks, one possible approach is to quantize weights layer by layer [32]. One can look at the statistical properties of all filters of a convolutional layer, i.e. choose minimum and maximum values across all filters, etc., and use the same scaling factor for all of the filters.

Alternatively, a more fine-grained approach is computing a separate scaling factor and quantizing each channel of a convolutional layer separately. A more extreme approach would be to perform sub-channelwise quantization where we quantize any group of parameters. A similar approach to sub-channelwise quantization can be seen in [58] where the authors quantize consecutive fixed-size blocks of input values. Sub-channelwise quantization and similar quantization methods offer a trade-off, however. Depending on the size of the groups, maintaining multiple scaling factors is complex and adds overheads in terms of computation and space.

Stochastic Quantization Usually, quantization functions are deterministic. The *Int(.)* function in equation 2.8 uses a deterministic rounding scheme like round-to-nearest. Some quantization functions, however, prefer to use stochastic rounding schemes [7, 65]. The general idea behind using stochastic quantization functions has been to achieve larger weight updates and allow neural networks to explore more in hopes of converging to a better accuracy.

Stochastic rounding schemes are also particularly useful for gradient quantization during backpropagation. It can be proven that stochastic rounding schemes have zero bias and quantized gradients are an unbiased approximation of unquantized gradients.

Theorem 2.1 *Stochastic Rounding schemes are unbiased.*

Theorem 2.2 *Stochastic gradient quantization produces an unbiased approximation of unquantized gradients.*

Stochastic quantization increases MSE relative to deterministic quantization functions like round-to-nearest (RDN). Also, there isn't a guarantee of unbiasedness of quantized activations and weights due to the non-linearity of activation functions. Therefore, stochastic quantization is more likely to degrade accuracy if used in forward passes [7].

Theorem 2.3 *The round-to-nearest rounding scheme has a lower mean squared error (MSE) than stochastic quantization.*

For the proofs of the theorems, see Appendix A.

Quantization Aware Training (QAT) Quantizing a trained model might perturb the trained weights such that the model might move away from its converged point. In order to achieve good accuracy it might be necessary to finetune the model. In general, this can be done in various ways such as Quantization Aware Training [46], Post-Training Quantization [46] or Zero Shot Quantization [5]. QAT retrains the model and requires access to training and validation datasets. In PTQ, quantization is applied to the trained model, therefore it is faster yet less accurate than QAT. ZSQ performs quantization without any access to training or validation datasets which makes it useful for Machine Learning as a Service platforms where users might not want to share their datasets. The method that is relevant to this thesis is QAT. QAT is an algorithm used to reduce the quantization noise. It models the quantization source during the training [46]. Therefore, as we mentioned, it requires access to training data and labels to retrain the model to be robust against quantization noise.

QAT is usually slow and resource-intensive as it might be necessary to retrain the model for many epochs before reaching the previous convergence accuracy. The forward and backward passes are performed on the quantized model. Then, after each backpropagation, the model parameters are requantized. It is important to store the parameters as floating point tensors as quantization functions are effectively step functions and therefore gradients will usually be zero. The quantization function can usually be approximated by using methods like Straight Through Estimator (STE) which essentially ignores the rounding function and replaces it with the identity function [3].

2.3 Pruning

In this section, we introduce neural network pruning, a compression method applied to neural networks to reduce memory size and bandwidth. Pruning, also known as sparsification, is the process of removing unnecessary parameters to reduce resource usage and speed up neural networks which help with deploying neural networks on constrained environments like embedded platforms and improving performance.

Since most neural networks are overparameterized, it is possible to eliminate significant amounts of the parameters without decreasing the accuracy too much [59]. In convolutional neural networks, for example, it is possible to eliminate around 70 to 95% of the parameters [19]. Neural network pruning can have vastly different granularities. We can either prune individual parameters or entire rows, columns, filters, and even layers. The first type of pruning is usually described as an unstructured pruning method while the rest are described as structured pruning [41] which we discuss next.

Neural networks can be pruned by either removing weights which are zero or close to being zero or replicated [41].

Unstructured Pruning Unstructured pruning is the removal of less important weights wherever they might be without showing any regard for their connections. Somewhat random removal of the weights means that pruned weights need to use sparse matrix and tensor representations as there is no guarantee that non-zero values would be grouped together. Sparse representations usually require extra information such as the indices of the non-zero values to be stored explicitly. This can require significant additional space if the sparsity of the parameters is not high.

As it is unlikely that parameters are packed densely, operations involving the sparse parameters are likely to be memory bound which can limit performance gains even though the number of computations might have decreased considerably. Unstructured sparsity also limits the use of SIMD operations. Therefore, unstructured pruning is usually useful at high amounts of sparsity as we would perform significantly less computations which would make up for the lack of SIMD instructions and lower arithmetic intensity.

Structured Pruning Structured pruning removes structured parts of the parameters ranging from entire rows, columns, filters, and layers. In other words, in structured pruning instead of storing individual non-zero elements, contiguous blocks of parameters are kept while rest of the parameters are pruned. This allows efficient execution on hardware optimized for dense computations [26].

Some examples of structured pruning involve Layer Pruning [20, 56] where entire Transformer blocks can be dropped to compress models. It has been suggested that around half of the layers can be dropped for 2x speed up [66]. Other examples include Head Pruning [62, 44] and Feed-Forward Neural Network (FNN) Pruning [52, 6] where individual

attention and feed-forward layers are pruned completely. Head Pruning has been shown to achieve a speed-up of up to 1.4x [40].

Apart from these coarse-grained structured pruning methods, there are also more fine-grained pruning methods like block pruning [35, 68]. In block pruning, parameters are divided into fixed-size blocks. Parameters in the same blocks are pruned together. Block-pruned parameters can use sparse representations similar to parameters pruned using unstructured pruning methods. However, they usually suffer less memory overhead, as instead of maintaining the positions of each non-zero parameter, we just have to maintain the positions of non-zero blocks. Additionally, block pruning is also more likely to be more computationally efficient than its unstructured counterparts as the parameters are densely packed which could enable the use of algorithms with higher arithmetic intensity as well as the use of SIMD instructions. However, it proved to be difficult to optimize models with block sparsity [67].

Sparse Representations As we mentioned previously, sparse tensors pruned using unstructured pruning or block pruning require special sparse representations. While most structured pruning methods maintain the general shape and structure of the parameters by removing entire structures like full columns or rows, unstructured pruning can remove any parameter. Here, by removing, we mean zeroing the parameter values. Because zeroes and non-zeroes are not intertwined, in structured pruning, we can easily exclude the pruned parameters without affecting the non-zero values. In unstructured pruning, however, we cannot simply remove the structure that contains the pruned parameters, i.e. remove a pruned filter without removing non-zero parameters. This is why unstructured pruning requires special representations to track the positions of all non-zero parameters.

For two-dimensional matrices, there have been a considerable amount of sparse representations developed both for unstructured and structured matrices. Unstructured sparse matrix representations include Dictionary of Keys (DoK) [18], List of Lists (LoL), Coordinate List (COO) [2], Compressed Sparse Row (CSR) [55] and Compressed Sparse Column (CSC) [55]. Dictionary of Keys, as its name suggests, is a dictionary that maps the (row, column) pair to its respective non-zero value. As the primary storage is a map, it might be difficult to iterate over the non-zero values in sorted order. To address this issue, one could use COO which is essentially DoK where tuples with three entries (row, column, value) are stored, usually, in sorted order. Alternatively, LoL stores one list per row where the list entries are (column, value) pairs, usually sorted by column. CSR and CSC are arguably the most used sparse matrix representations along with COO. CSR maintains three lists. One list stores the non-zero values in sorted order where the elements are sorted by their (row, column) values. The second list stores the respective column value of the non-zero element. There is a one-to-one correspondence between the elements of the value list and the column list. Finally, the third list stores (# of rows + 1) elements where the ith element indicates the start of the ith row in the column and value lists. CSC representation is almost identical to CSR, however, this time the row list corresponds to the exact row of the non-zero values whereas the column list indicates the start of the ith column in the row and value lists.

Tensors are, in effect, generalizations of matrices. Therefore, sparse tensor representations are generally generalizations of sparse matrix representations. The generalization of CSR/CSC format is called compressed sparse fiber (CSF) representation [57]. CSF representation is a hierarchical sparse format that can be used to represent tensors with arbitrary amounts of dimensions/modes. Similar to CSR/CSC, CSF maintains arrays to address the start location of non-zero values in the next dimension. Visually speaking, CSF essentially forms a tree. Each level represents a dimension. Each non-zero value is a path from the root to its corresponding leaf [39]. Owing to its hierarchical, tree-like structure, accessing and

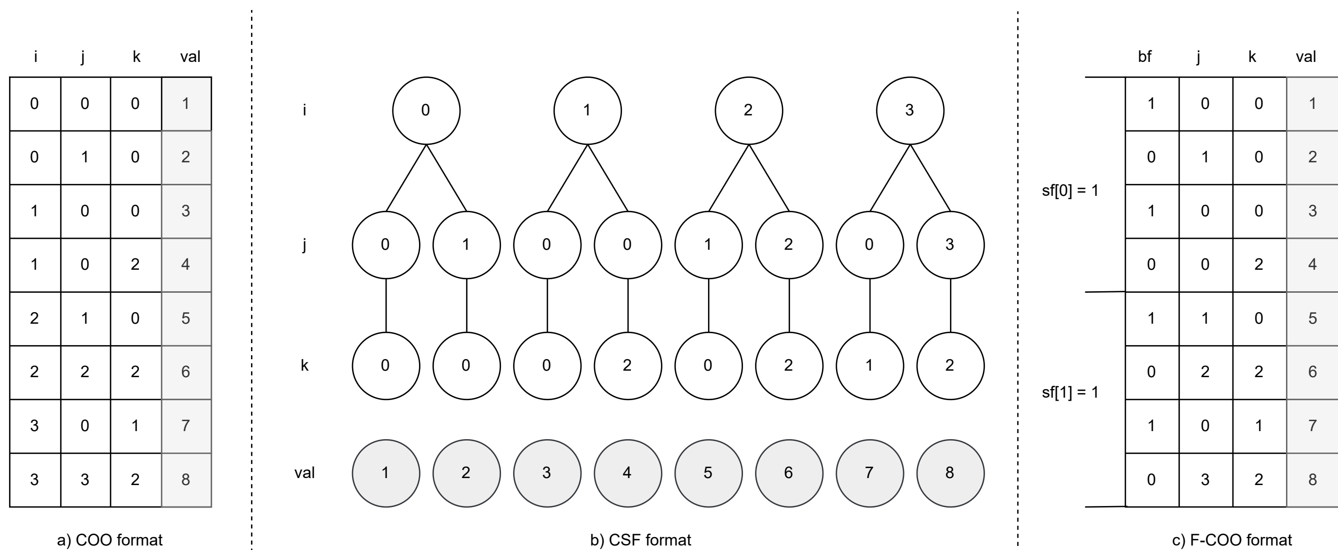


Figure 2.2: Visual representation of a three-dimensional sparse tensor using COO, CSF, and F-COO formats. The figure is modeled after Figure 2 in [39].

traversing non-zero elements in an order different from the hierarchy might incur some performance costs. One might overcome this issue by storing the same sparse tensor in different orientations of the dimensions, however, as the number of dimensions increases, this strategy becomes more costly than it is worth.

There are also COO inspired sparse format generalizations for tensors. Two such formats are flagged COO (F-COO) [42] and hierarchical COO (HiCOO) [39]. F-COO stores the indices of dimensions that would be required by arithmetic operations. All the other dimensions that will not be used are replaced by two 1-bit flags, bit-flag (bf) and start-flag (sf). On a high level, these flags are used to indicate whether changes in computations occurred which led to a use in the previously unused dimensions. Figure 2.2 shows the visualizations of the COO, CSF, and F-COO formats.

HiCOO format, depicted in Figure 2.3, is like a combination of CSF and COO. Tensors are divided into fixed-sized blocks. For an N-dimensional tensor, it uses N variables to index the blocks and another N variables to index the position of elements within the blocks. Additionally, it stores pointers to the beginning of each block. Only the non-zero values of the tensors are stored. Blocking enables more compact storage of the non-zero elements, as we can use fewer bits to store the element and block indices, compared to full tensor. Additionally, because HiCOO is modeled after COO format, it does not make any assumptions about the orientations of the dimensions (mode generic). Thus, accessing/traversing the non-zero elements in any arbitrary dimension orientations has the performance. However, HiCOO has a limitation where it expects the blocks to be ”cubical”, that is, each block dimension should have the same size.

Pruning Methods Because of their computational benefits, a renewed interest in pruning led to the development of a variety of pruning methods. For model training, there were already methods that implicitly pruned unnecessary parameters. Using variational dropout or L0 regularization, for example, either removed the neurons in their entirety or effectively zeroed parameters already. More recently, sparsification frameworks like Neural Magic’s SparseML [15] have incorporated pruning methods like Gradual Magnitude Pruning (GMP) [23, 24, 70] and Alternative Compress/DeCompress (AC/DC) Pruning [51] to achieve 90-95% sparsity with minimal accuracy loss.

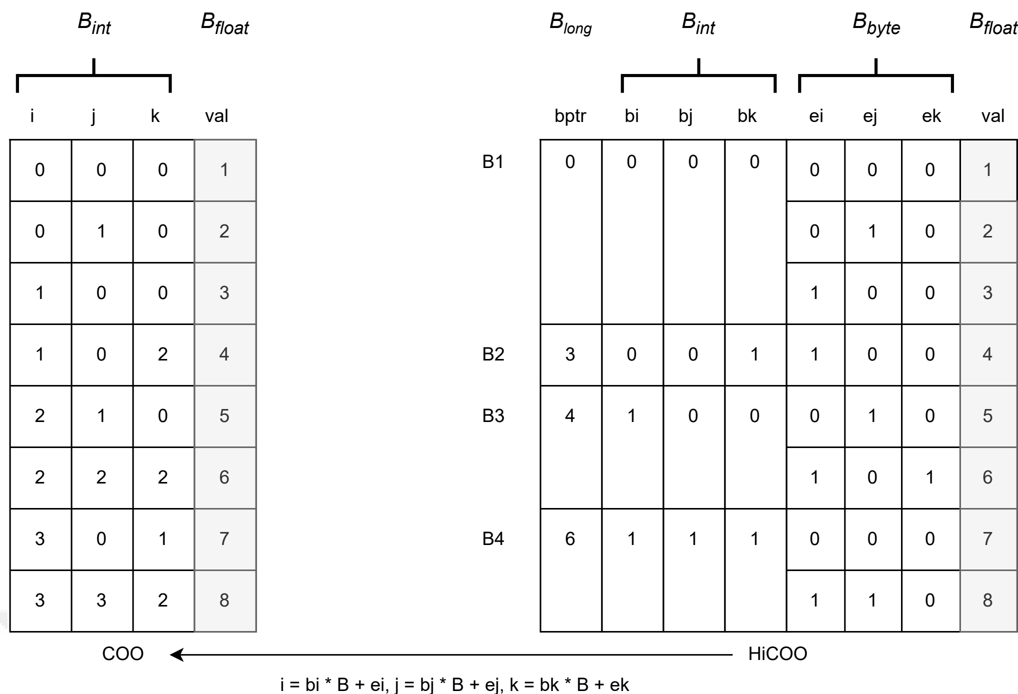


Figure 2.3: The conversion from COO to HiCOO format. HiCOO uses blocks of size 2x2x2 (B=2). The sparse tensor in this figure is the same one in Figure 2.2. The figure is modeled after Figure 3 in [39].

The main idea behind GMP is to select a threshold and prune all parameters that have an absolute value less than the threshold. The base algorithm introduced in [24] follows three stages. The first step is to train a neural network to learn its "important" connections. The second step is to remove the "unimportant" connections using a threshold like we have mentioned. Finally, the model is retrained with the pruned parameters to improve accuracy. These three steps are repeated multiple times during training. During model retraining in the final step, it is better to use the parameters that were not pruned in the second step instead of reinitializing new parameters. Other caveats include using L2 regularization instead of L1 and adjusting the number of neurons dropped during training due to dropout to account for the connections and neurons already pruned. L2 regularization produces better results after training [24].

AC/DC pruning effectively trains two networks. During training, it switches back and forth between dense and sparse model training phases. AC/DC pruning uses a standard training loop one would use to train regular NNs. The first few epochs are used as a warm-up to train a dense model. After that, the training loop alternates between sparse and dense phases for a fixed number of epochs. During the sparse phases, all the parameters except the top k (in absolute value) are pruned for a hyperparameter k. A mask is used to ensure the pruned weights remain zeroed for the rest of the sparse phase. During dense phases, the mask is set to a tensor of all ones.

Chapter 3

Quantization and Sparse Kernel Implementations

In this chapter, we discuss the implementation details of high-performance, quantized, sparse kernels used in the forward and backward passes of the backpropagation algorithm. We start our chapter by introducing the quantization techniques we used in the forward and backward passes. Then, we move on to the implementations of forward and backward passes for sparse weight tensors in linear and convolutional layers. Here, we also discuss the data models we used for sparse linear and convolutional weights. Finally, we introduce multi-threaded implementations of all of the kernels for improved performance.

3.1 Quantization Kernels

In this section, we discuss the quantized data representation and describe the implementations of the quantization techniques we used: Statistics aware weight binning (SAWB) [8, 9] and dithered quantization [65]. The former is a symmetric, deterministic, integer quantization method that we use during the forward passes. The latter is a stochastic integer quantization method, originally designed as a floating point quantization method, we use during the backward passes.

3.1.1 Quantized Representation

Our library implements INT8 quantization functions. Hardware support for low-precision floating point numbers is not widespread. As of writing this, AVX instructions can only support 16-bit half-precision floating point representations [14]. There is currently no IEEE standard for floating point numbers with less than 16 bits [27]. To the best of our knowledge, the current support for FP8 data types is for GPUs only. In 2022, NVIDIA, AMD, and Intel jointly published a whitepaper for FP8 formats [45]. In the same year, NVIDIA released the H100 Tensor Core GPU [12] and its 40 series GPUs [11] which are capable of supporting F8 data types.

We assume the quantization inputs to be 32-bit floating point numbers. In return, the quantization functions output 8-bit signed integer tensors and floating point scales. Scales are later used for dequantization where we revert 8-bit integers back into 32-bit floating point numbers. 32-bit floating point numbers are initially quantized into 32-bit integer numbers which can be represented with only 8 bits. We then pack the 32-bit integers directly in 8-bit tensors.

During the quantization and dequantization processes, it is important to efficiently pack 32-bit integers into 8-bit tensors and unpack 8-bit integers into 32-bit tensors. In our library,

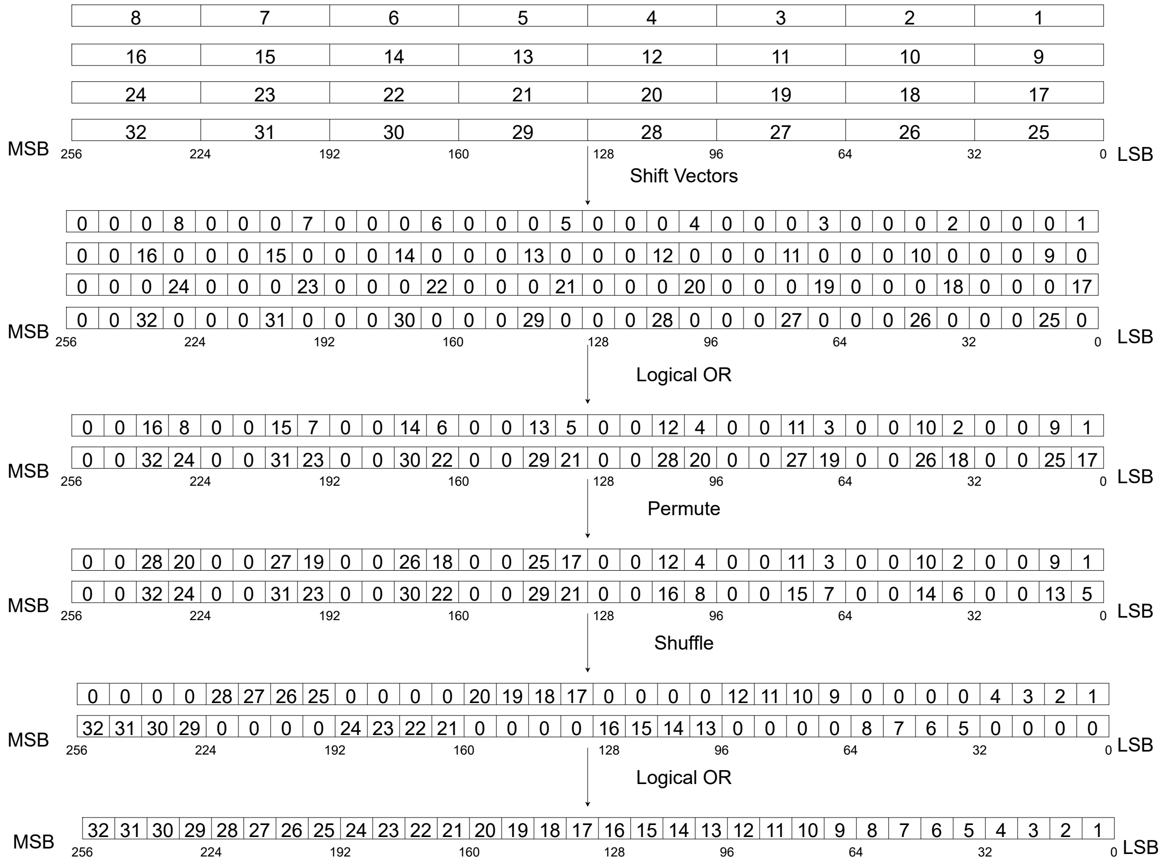


Figure 3.1: Step-by-step visualization of the packing process. Every array represents a 256-bit integer vector.

we use packing and unpacking routines introduced in [58].

Packing The packing procedure works on unpacked, 32-bit quantized integer vectors and outputs packed, 8-bit quantized integer vectors. The procedure starts by shifting each unpacked integer element by 24 bits to the left. Since all of our quantized values are stored in the 8 least significant bits, this operation eliminates all of the excess sign bits.

Next, we perform a logical right shift to move the integers into the desired positions. As it can be seen from Figure 3.1 our procedure packs 32 integers into a single 256-bit vector. Therefore, we first combine vectors 0 & 1 and vectors 2 & 3 with each other using a bitwise or operation. With this operation, we have combined two 8-bit integers in 32 bit slots. These two integers are 8 elements apart from each other in the consecutive vectors.

Then, we permute the two combined vectors to obtain two more vectors. The first vector contains the concatenation of the lower halves of the previous two vectors. Conversely, the second vector contains the upper halves. In the penultimate step, we shuffle the lower and upper half vectors, such that we place groups of 4 consecutive elements in the correct positions. Finally, we combine the shuffled halves with a bitwise or operation to obtain a single packed vector. Appendix B shows the pseudocode for the packing algorithm.

Unpacking The unpacking process is the inverse operation of the packing process. The unpacking routine follows the same operations as the packing routine in the reverse order to convert a 256-bit vector filled with 32 8-bit packed integers into 4 256-bit vectors with 8 32-bit unpacked integers.

Figure 3.2 shows how we unpack quantized vectors step-by-step. We start unpacking by

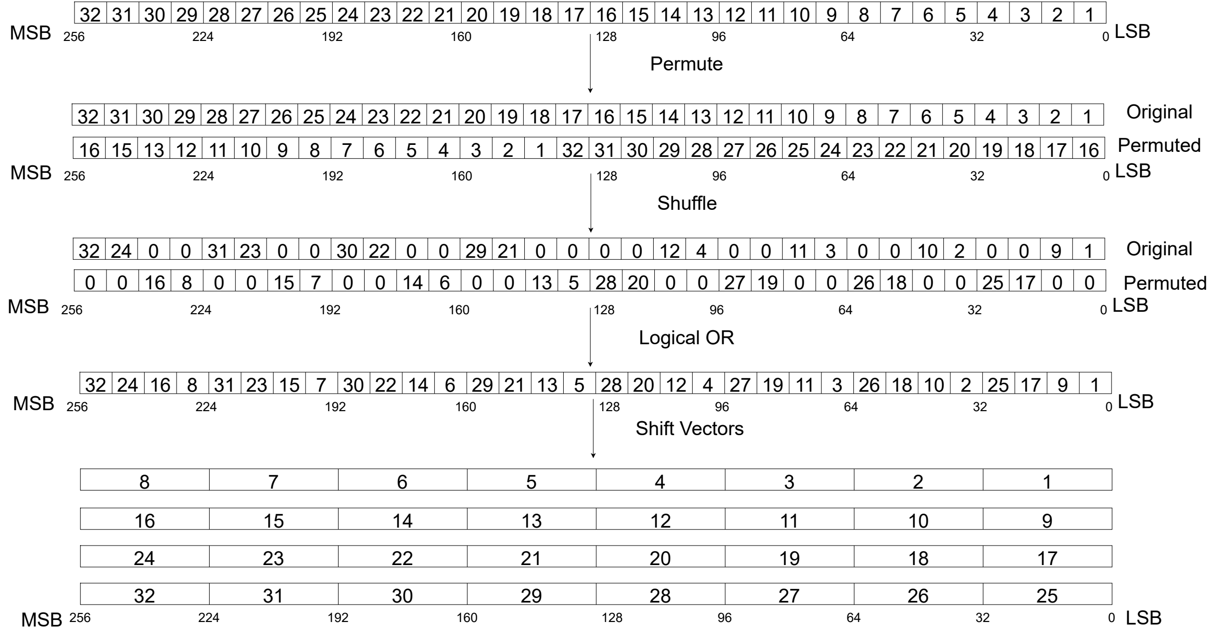


Figure 3.2: Step-by-step visualization of the unpacking process. Every array represents a 256-bit integer vector.

creating an auxiliary vector where we permute the original vector to switch the positions of the lower half with the upper half. Next, we perform two shuffle operations on the original and auxiliary vectors respectively.

In the final state of the packed 256-bit vector, we want to store the elements such that each consecutive group of four integers is eight positions apart. Therefore, the first shuffle creates a vector where we place the lower halves of the four consecutive elements. Similarly, the second shuffle creates a vector with the upper halves placed in the correct positions. The rest of the positions are filled with zeros. After shuffling, we combine these two vectors with a logical or operation.

Finally, we first perform a logical shift to the left and then an arithmetic shift to the right to obtain 4 256-bit unpacked vectors.

3.1.2 SAWB Quantization

SAWB quantization is the first of the two quantization techniques we have implemented for our library. As mentioned at the beginning of the chapter, it is a deterministic, symmetric integer quantization technique first introduced in [8, 9]. We use it during the forward passes in neural networks. To achieve high performance, the algorithm is implemented using SIMD instructions.

Rounding schemes have an important effect on the accuracy of the trained model. It can be proven that round-to-nearest (RDN) rounding has better mean-squared-error (MSE) than stochastic rounding [7]. Therefore, it is better to use deterministic rounding schemes as opposed to stochastic ones. Stochastic rounding is normally unbiased (Appendix A), however, due to non-linearity of the loss and activation functions it still introduces bias in the forward passes as well, worsening the quantization error [7].

Quantization The quantization algorithm is straightforward. It takes the tensor to be quantized A, the value range for the quantized array Q_u & Q_l, and two hyperparameters c_1 & c_2 as inputs. It outputs the quantized array QA and a scaling factor.

We first compute the scaling factor according to the following equations

$$clip = c_1 \times \sqrt{E(A^2)} - c_2 \times (E(|A|)) \tag{3.1}$$

$$scale = 2 \times \frac{clip}{Q_u - Q_l} \tag{3.2}$$

where $E(|A|)$ & $E(A^2)$ denote the first and second moments of tensor A. Algorithm 1 shows the pseudocode for SIMD implementation of the scale.

Algorithm 1 Algorithm to compute SAWB scale

Input: A, Q_u, Q_l, c_1, c_2
Output: $scale$

1: $vacc_sq_mean \leftarrow$ **vbcast**(0)
2: $vacc_abs_mean \leftarrow$ **vbcast**(0)
3: $N \leftarrow n$
4: **for** $i \leftarrow 0$ **to** $size(A) - 15$ **with** i += 16 **do**
5: $\quad val_A \leftarrow$ **vload**$(A)_i$
6: $\quad val_abs_A \leftarrow$ **vand**$(val_A,$ **vbroadcast**$(0x7FFFFFFFU))$
7: $\quad vacc_sq_mean \leftarrow$ **vfmadd**$(val_A, val_A, vacc_sq_mean)$
8: $\quad vacc_abs_mean \leftarrow$ **vadd**$(val_abs_A, vacc_abs_mean)$
9: **end for**
10: $acc_sq_mean \leftarrow$ **vhadd**$(vacc_sq_mean)/size(A)$
11: $acc_abs_mean \leftarrow$ **vhadd**$(vacc_abs_mean)/size(A)$
12: $clip \leftarrow c_1 \times \sqrt{acc_sq_mean} - c_2 \times (acc_abs_mean)$
13: $scale \leftarrow 2 \times \frac{clip}{Q_u - Q_l}$
14: **return** scale

SAWB is a quantization technique originally designed to quantize the weights in neural networks. During the computation of the scale, it uses the first moment $E(|A|)$ of the input A to obtain a representative value of the input. It uses the second moment $E(A^2)$ of the input to capture the shape of the distribution of the input [9]. The shapes of the distribution for weights change during the training process when the backward pass happens. We want to find a scale that minimizes the quantization error for a large variety of distributions in order to account for the change in weight distributions during training. In [9], the authors consider six distributions: Gaussian, uniform, Laplace, logistic, triangle, and von Mises. They plot $\sqrt{E(A^2)}/E(|A|)$ against $scale/E(|A|)$ using the six distributions and compute the linear regression of the optimal scales for these distributions to compute the coefficients c_1 & c_2.

After computing the scale, we use it to compute the quantized output QA. The values are quantized according to the formula:

$$QA = RDN(min(max(Q_l, \frac{A}{scale}), Q_u)), \tag{3.3}$$

where we ensure the quantized values are within the range $[Q_l, Q_u]$ and they are deterministically rounded to the nearest integer. By choosing Q_l & Q_u accordingly, we can control how many bits are needed for the quantized integers. As we implement 8-bit quantization techniques, we choose Q_l to be -128 & Q_u to be 127.

Algorithm 2 shows the pseudocode for the full SAWB quantization process. We compute the quantized output in two passes. The first pass computes the scale. The second pass

quantizes the input using the scale. We assume the input tensor is contiguous and addressed by a 1D pointer. The quantization loop is unrolled once to iterate over 32 consecutive elements of A since our packing algorithm packs 32 unpacked integers. To quantize input, we first compute the reciprocal of the scale and then multiply the input with the reciprocal. This is because multiplication operation is more efficient than division operation. Then we clip the resulting number, so it is in the interval $[Q_l, Q_u]$. Finally, we round the values to the nearest integer.

Algorithm 2 SAWB Quantization

Input: A, Q_u, Q_l, c_1, c_2
Output: QA

1: $scale \leftarrow$ **compute_scale**(A, Q_u, Q_l, c_1, c_2)
2: $vrcp_scale \leftarrow$ **vbcast**$(1.0/scale)$
3: $vlower_bound \leftarrow$ **vbcast**(Q_l)
4: $vupper_bound \leftarrow$ **vbcast**(Q_u)
5: **for** $i \leftarrow 0$ **to** $size(A) - 31$ **with** i += 32 **do**
6: $val_f_{A0} \leftarrow$ **vload**$(A)_i$
7: $val_f_{A1} \leftarrow$ **vload**$(A)_{i+16}$
8: $val_f_{A0} \leftarrow$ **vmul**$(val_f_{A0}, vrcp_scale)$
9: $val_f_{A1} \leftarrow$ **vmul**$(val_f_{A1}, vrcp_scale)$
10: $val_f_{A0} \leftarrow$ **vmin**(**vmax**$(val_f_{A0}, vlower_bound), vupper_bound)$
11: $val_f_{A1} \leftarrow$ **vmin**(**vmax**$(val_f_{A1}, vlower_bound), vupper_bound)$
12: $val_i_{A0} \leftarrow$ **vcvtps_epi32**(val_f_{A0})
13: $val_i_{A1} \leftarrow$ **vcvtps_epi32**(val_f_{A1})
14: $vpacked \leftarrow$ **pack32**(val_i_{A0}, val_i_{A1})
15: **vstore**$(QA_i, vpacked)$
16: **end for**
17: **return** QA

Dequantization Dequantization is the inverse of the quantization process. Therefore, in order to recover the dequantized array A_{dq}, the approximation to the original input, we reverse our operations from the quantization process. The dequantization formula for the ith element of the quantized tensor is as follows:

$$A_{dq}[i] = scale \times QA[i], \tag{3.4}$$

We simply multiply the quantized value with the scale. However, in order to get to that stage, we first need to do some preprocessing. Algorithm 3 shows the pseudocode for dequantizing 32 elements in QA.

We first unpack the quantized integers. Next, we convert them into floating-point numbers. Finally, we multiply the unpacked floats with the scale to obtain the dequantized values.

3.1.3 Dithered Quantization

Dithered quantization, also known as non-subtractive dithered quantization, is the second of the two quantization techniques we implemented for our library. Unlike SAWB quantization, dithered quantization is a stochastic quantization technique. It can be shown that unbiased stochastic quantization functions can be used to compute unbiased estimators of the actual gradients which helps us reduce quantization error (Appendix A). Therefore, we use dithered quantization to quantize gradients in the backpropagation algorithm.

Algorithm 3 SAWB Dequantization

Input: $A, i, vscale$

1: $q_vvals \leftarrow \mathbf{vload}(A)_i$
2: $unpacked_0, unpacked_1, unpacked_2, unpacked_3 \leftarrow \mathbf{unpack32}(q_vvals)$
3: $u_q_vals_0 \leftarrow \mathbf{vcvtepi32_ps}(unpacked_0)$
4: $u_q_vals_1 \leftarrow \mathbf{vcvtepi32_ps}(unpacked_1)$
5: $u_q_vals_2 \leftarrow \mathbf{vcvtepi32_ps}(unpacked_2)$
6: $u_q_vals_3 \leftarrow \mathbf{vcvtepi32_ps}(unpacked_3)$
7: $dq_vals_0 \leftarrow \mathbf{vmul}(vscale, u_q_vals_0)$
8: $dq_vals_1 \leftarrow \mathbf{vmul}(vscale, u_q_vals_1)$
9: $dq_vals_2 \leftarrow \mathbf{vmul}(vscale, u_q_vals_2)$
10: $dq_vals_3 \leftarrow \mathbf{vmul}(vscale, u_q_vals_3)$

Dithered quantization was originally designed as a floating point quantization technique, meaning that, unlike SAWB, it outputs low bit-width floating point numbers after the quantization process. As we have mentioned previously, there is no standard or widespread hardware support for floating point numbers with less than 16 bits [27]. Since our library implements INT8 quantization kernels, we had to adjust our dithered quantization implementation to account for that limitation. The definition for the dithered quantization function is defined in [65] as

$$\begin{aligned} QA &= Q_\Delta(A+v), \\ &= \Delta \left\lfloor \frac{A+v}{\Delta} + \frac{1}{2} \right\rfloor . \end{aligned} \tag{3.5}$$

Here, $A \in \mathbb{R}$ is our input and $v \sim U(-\frac{\Delta}{2}, \frac{\Delta}{2})$ is uniform noise in the open interval $(-\frac{\Delta}{2}, \frac{\Delta}{2})$. Δ is the scaling factor. Since, we implement INT8 quantization functions, in our implementation, the floor function is part of the quantization function. Multiplying the floored value with the scaling factor results in dequantizing the quantized tensors. We clip the floored value to ensure the quantized values fit in 8-bit integers. In the original paper, the authors choose $\Delta = s\sigma$ where $s \in \mathbb{R}$ is the scaling factor and σ is the standard deviation, so, we choose Δ to be equal to that too.

Random Number Generation. Before we move on to the quantization and the dequantization processes, we first want to discuss the random number generation process in our library. Random number generation is pivotal to achieving good accuracy as it enables us to implement unbiased quantization functions.

Our library initially used PyTorch to generate uniform random noise, however, in order to achieve better performance, we later switched to AVX512 implementation of xorshift128+ [38]. As the name might imply, xorshift128+ is a random number generator (RNG) in the family of XORShift RNGs [43]. It generates sequences of 128 bit random numbers. It doesn't have very strong statistical properties, however, it is easy to parallelize.

The algorithm is simple. We start with a 128-bit randomly generated seed split into two 64-bit variables ($uint64_t$). Since we want to fill 512-bit vectors, we initialize two $uint64_t$ vectors of size 8. Each element in the vector is initialized to be 2^{64} **next()** calls apart in order to avoid overlapping randomly generated segments in the 512-bit arrays. We use a special jump function to compute 2^{64} steps ahead.

Algorithm 4 shows the implementation of **next()**. We use a vectorized version of algorithm 4 and previous value of the random numbers to generate new random values.

Algorithm 4 Xorshift128+ Generate Next Random Value

Input: $seed_0, seed_1$
Output: rnd_0, rnd_1
1: $tmp_seed_1 \leftarrow seed_0$
2: $tmp_seed_0 \leftarrow seed_1$
3: $rnd_0 \leftarrow tmp_seed_0$
4: $tmp_seed_1 \leftarrow tmp_seed_1 \oplus (tmp_seed_1 \ll 23)$
5: $rnd_1 \leftarrow tmp_seed_1 \oplus tmp_seed_0 \oplus$
$(tmp_seed_1 \gg 18) \oplus (tmp_seed_0 \gg 5)$
6: **return** rnd_0, rnd_1

Instead of calling the **next()** function 2^{64} times, we efficiently compute the non-overlapping random segments using algorithm 5. As you can see, only 128 calls to **next()** is sufficient.

Algorithm 5 Jump Algorithm

Input: $seed_0, seed_1$
Output: out_0, out_1
1: $JUMP[2] \leftarrow \{0x8a5cd789635d2dff, 0x121fd2155c472f96\}$
2: $out_0 \leftarrow 0$
3: $out_1 \leftarrow 0$
4: **for** $i \leftarrow 0$ **to** $(\textbf{sizeof}(JUMP)/\textbf{sizeof}(*JUMP))$ **with** i++ **do**
5: **for** $b \leftarrow 0$ **to** 64 **with** b++ **do**
6: **if** $JUMP[i]\&1ULL \ll b$ **then**
7: $out_0 \leftarrow out_0 \oplus seed_0$
8: $out_1 \leftarrow out_1 \oplus seed_1$
9: **end if**
10: $seed_0, seed_1 \leftarrow \textbf{next}(seed_0, seed_1)$
11: **end for**
12: **end for**
13: **return** out_0, out_1

Finally, algorithm 6 shows how we initialize the initial state for AVX512 vectors using the jump algorithm.

Algorithm 6 Xorshift128+ Init Algorithm

Input: $seed_0, seed_1$
Output: key_0, key_1
1: $S0[8] \leftarrow \{seed_0\}$
2: $S1[8] \leftarrow \{seed_1\}$
3: **for** $i \leftarrow 1$ **to** 8 **with** i++ **do**
4: $S0[i+1], S1[i+1] \leftarrow \textbf{jump}(*(S0+i), *(S1+i))$
5: **end for**
6: $\textbf{vstore}(S0, key_0)$
7: $\textbf{vstore}(S1, key_1)$
8: **return** key_0, key_1

As mentioned before, while xorshift128+ RNG is not statistically robust, i.e. it fails statistical tests such as Big Crush [37], we found that training accuracy did not degrade with respect to PyTorch's RNGs while increasing performance, which was sufficient enough to

justify our choice.

Quantization To quantize input tensors, we first compute the quantization scale similarly to SAWB quantization. Recall that our quantization scale $\Delta = s\sigma$ requires two values. The first one is $s \in \mathbb{R}$, the scaling factor. The second one is the standard deviation. The scaling factor is a hyperparameter chosen by us. Unlike SAWB, dithered quantization does not have a deterministic way of specifying the quantization intervals, that is, we can not precisely control the number of bits the quantized integers will have without clipping. We observed that, as s becomes smaller, the range of the quantization interval increases, which makes sense as s is part of the denominator in equation 3.5. However, as the range grows it will eventually not fit in 8-bit integers. We found that using $s = 0.5$ increased the bit width the most while minimizing clipped values during training, thus achieving lower quantization error.

The naive way of computing the standard deviation requires us to pass over the data twice, increasing the data transfers from memory and thus lowering the benefits of quantization. To compute the standard deviation efficiently we rearrange the formula as

$$\sigma = \sqrt{\frac{1}{N} \times \sum (x_i - \mu)^2} = \sqrt{\frac{\sum (x_i)^2}{N} - \mu^2}. \tag{3.6}$$

After this rearrangement, it is easy to see that we can compute the standard deviation in a single pass by computing the sum of squares and the mean of the input. Algorithm 7 shows the AVX implementation of the computations for the quantization scale.

Algorithm 7 Compute Dithered Quantization Scale Algorithm

Input: $A, scaling_factor$
Output: $scale$
1: $v_acc_sq_sum \leftarrow$ **vbcast**(0)
2: $v_acc_mean \leftarrow$ **vbcast**(0)
3: **for** $i \leftarrow 0$ **to** size(A) - 15 **with** i += 16 **do**
4: $val_vec \leftarrow$ **vload**$(A)_i$
5: $v_acc_sq_sum \leftarrow$ **vfmadd**$(val_vec, val_vec, v_acc_sq_sum)$
6: $v_acc_mean \leftarrow$ **vadd**(val_vec, v_acc_mean)
7: **end for**
8: $sq_sum \leftarrow$ **vhadd**$(v_acc_sq_sum)$
9: $mean \leftarrow$ **vhadd**$(v_acc_mean)/size(A)$
10: $scale \leftarrow scaling_factor \times \sqrt{(sq_sum/size(A)) - mean^2}$
11: **return** scale

After computing the scale, we can easily quantize the values using the scale and the random noise we generate through the xorshift128+ RNG. We can quantize the input as state in equation 3.5. Algorithm 8 shows the SIMD implementation of the dithered quantization process. We first generate the scale using algorithm 7. Then, for each input vector, we create random noise using the vectorized version of algorithm 4. We add the random noise to input vectors to ensure the unbiasedness of quantized gradients. We divide this value by the scale and add a constant of 0.5. Finally, we take the floor of the quantized value to convert the values into 8-bit integers and pack them.

Dequantization The dequantization process in dithered quantization is identical to the dequantization process in SAWB quantization. For dequantization, the only information

Algorithm 8 Dithered Quantization

Input: $A, scaling_factor, seed_0, seed_1$
Output: QA
1: $scale \leftarrow$ **computeScale**$(A, scaling_factor)$
2: $vrcpscale \leftarrow$ **vbcast**$(1.0/scale)$
3: **for** $i \leftarrow 0$ **to** $size(A) - 31$ **with** $i+ = 32$ **do**
4: $val_f_{A0} \leftarrow$ **vload**$(A)_i$
5: $val_f_{A1} \leftarrow$ **vload**$(A)_{i+16}$
6: $rnd_xor_0 \leftarrow$ **next_rnd**$(seed_0, seed_1)$ ▷ Compute random noise
7: $rnd_i8_0 \leftarrow$ **vand**$(rnd_xor_0,$ **vbroadcast**$(0x7f7f7f7f))$
8: $rnd_i8_1 \leftarrow$ **vslli**$(rnd_i8_0, 8)$
9: $rnd_f8_0 \leftarrow$ **vcvtepi32_ps**(rnd_i8_0)
10: $rnd_f8_1 \leftarrow$ **vcvtepi32_ps**(rnd_i8_1)
11: $rnd_scaled_0 \leftarrow$ **vfmsub**$(rnd_f8_0,$ **vbcast**$(2^{-31}),$ **vbcast**$(0.5))$ ▷ Rescale
12: $rnd_scaled_1 \leftarrow$ **vfmsub**$(rnd_f8_1,$ **vbcast**$(2^{-31}),$ **vbcast**$(0.5))$
13: $val_f_{A0} \leftarrow$ **vfmadd**$(vscale, rnd_scaled_0, val_f_{A0})$ ▷ Add random noise
14: $val_f_{A1} \leftarrow$ **vfmadd**$(vscale, rnd_scaled_1, val_f_{A1})$
15: $val_f_{A0} \leftarrow$ **vfmadd**$(val_f_{A0}, vrcpscale,$ **vbcast**$(0.5))$
16: $val_f_{A1} \leftarrow$ **vfmadd**$(val_f_{A1}, vrcpscale,$ **vbcast**$(0.5))$
17: $val_i_{A0} \leftarrow$ **vcvtps_epi32**$($**vfloor**$(val_f_{A0}))$
18: $val_i_{A1} \leftarrow$ **vcvtps_epi32**$($**vfloor**$(val_f_{A1}))$
19: $vpacked \leftarrow$ **pack32**(val_i_{A0}, val_i_{A1})
20: **vstore**$(QA_i, vpacked)$
21: **end for**
22: **return** QA

we need is the scale Δ we computed during quantization. To dequantize, we backtrack our computations during quantization by first unpacking the packed 8-bit integers, then multiplying the unpacked 32-bit integers with the scale. After that, we can store the dequantized values in floating point tensors.

3.1.4 Quantization Granularity

Quantization granularity refers to breaking down the quantization input into smaller subsets and quantizing the subsets separately. The idea is to reduce the quantization noise by introducing an increased number of quantization scales. These scales are more representative of the subset they belong to. Ideally, in the worst-case scenario, the overall quantization noise is no worse than the case where there is only a single group.

In our library, we only use a single group. We haven't noticed any meaningful accuracy gain during training when we introduced grouping. Moreover, the quantization techniques we use have special edge cases where grouping becomes complicated. For example, when all the elements in the group have the same value, dithered quantization fails because the scale becomes zero. However, addressing these types of edge cases introduced performance overhead, so, we decided not to use multiple groups.

3.2 Sparse Propagation Kernels

In this section, we discuss the sparse representation of weights in the linear and convolutional layers of sparse neural networks. Furthermore, we describe fast vectorized implementations of the forward and backward passes of these layers for quantized activations,

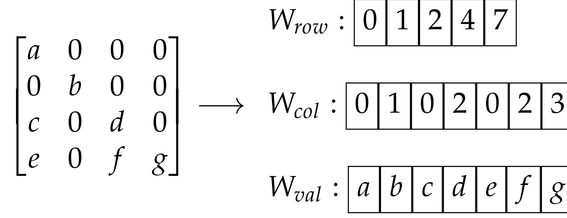


Figure 3.3: CSR representation of a sparse matrix

gradients, and sparse weights. Our representations and algorithms are intended to exploit the unstructured sparsity of models to achieve performance improvements and reduce training time. Most of the work in this section builds on top of [47] to speed up model training by quantizing sparse models.

3.2.1 Sparse Data Model for Linear Layers

Our library uses the compressed sparse row (CSR) format to represent sparse weights W in linear layers. Sparse representation uses three arrays, W_{row}, W_{col}, W_{val} to hold all of the nonzero values. W_{val} stores the nonzero values in the matrix. W_{col} stores the corresponding column of the nonzero value. W_{row} has number of rows + 1 entries. It encodes the starting position of the i-th row in the W_{col} and W_{val} arrays. In our implementation we use a third-party library, SciPy [61], to convert sparse matrices into CSR format. Fig. 3.3 shows an example of the CSR format of a sparse matrix.

3.2.2 Sparse Data Model for Convolutional Layers

In section 2.3 we discussed a variety of sparse formats for multi-dimensional tensors. To represent sparse weights in convolutional layers, we use a representation similar to CSR, introduced in [47]. For this layer, we maintain 5 arrays, W_{oc}, W_{ic}, W_x, W_y & W_{val}. W_{val} holds the quantized, non-zero values of the sparse convolutional weights. Similar to W_{col} in linear layers, W_x and W_y hold the exact position of the weight in its respective filter. W_{oc} & W_{ic} are analogous to W_{row} in sparse matrices. W_{oc} has (no. of output channels + 1) elements. W_{oc} encodes the starting position of the ith output channel in W_{ic} array. W_{ic} has ((no. of input channels + 1) $\times$ no. of output channels) elements. W_{ic} together with W_{oc} encodes the start of a specific channel in the W_x, W_y & W_{val} arrays using the following equation:

$$\begin{aligned} start_index_inclusive &= W_{oc}[oc] + W_{ic}[(IC+1)*oc+ic] \\ end_index_exclusive &= W_{oc}[oc] + W_{ic}[(IC+1)*oc+ic+1] \end{aligned} \tag{3.7}$$

Fig. 3.4 shows an example of the CSR-like format for sparse convolutional weights.

3.2.3 Linear Forward Propagation

The forward pass of fully connected layers is a general matrix-matrix multiplication (GeMM) algorithm. Let Q_A be the quantized activation tensor with scale s_A and Q_W be the quantized weights with scale s_W. Then, the output O of the layer f can be written as follows:

$$O = f(Q_A; Q_W) = s_A.s_W.Q_A Q_W \tag{3.8}$$

In the dense version, the input Q_A has dimensions B $\times$ M, weights Q_W have dimensions M $\times$ N and the output O has dimensions B $\times$ N where B is the batch size.

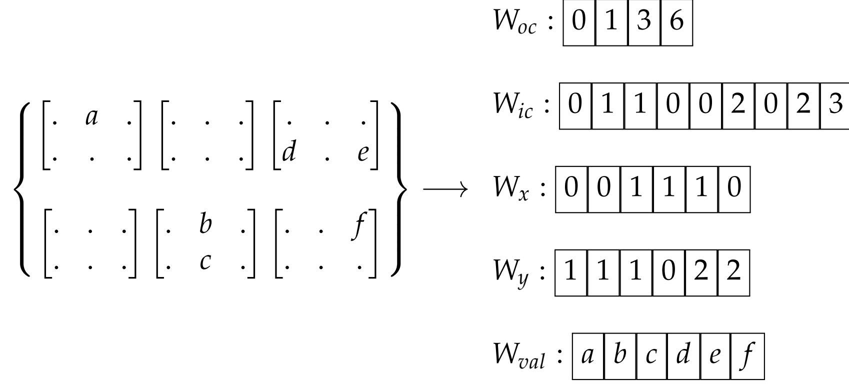


Figure 3.4: CSR-like representation of sparse weights in convolutional layers. Dense weight has the dimensions (OC, IC, X, Y) = (3, 2, 2, 3). Dots represent zeros. The figure is inspired by figures in [47].

Since the quantized weights are also sparse, we use the CSR representation introduced in subsection 3.2.1 to compute the layer output. Thus, we implement Sparse Matrix-Matrix Multiplication to compute equation 3.8 in algorithm 9. In the innermost loop, we iterate over dimension B of input to ensure we can use vectorized instructions even when the width of the network is small or sparsity is very high. However, this requires us to use relatively large batch sizes of 64 or more. Before we call the forward algorithm, we take the transpose of the activation to have streaming access over the elements of Q_A.

Algorithm 9 Quantized Sparse Forward Algorithm for Linear Layers

1: **for** $i \leftarrow 0$ **to** N **with** i++ **do**
2:   **for** $j \leftarrow W_idx_N[i]$ **to** W_idx_N[i + 1] **with** j++ **do**
3:     $w_mul_qa_scale_vec \leftarrow$ **vbroadcast**$(W_{val}[j] * s_W * s_A)$
4:     **for** $b \leftarrow 0$ **to** B - 63 **with** b += 64 **do**
5:       $o_0 \leftarrow$ **vload**$(O + i * B + b)$
6:       ... ▷ Repeat for the next 48 elements
7:       $qa_p \leftarrow$ **vload**$(QA + W_idx_M[j] * B + b)$
8:       $qa_{up0}, qa_{up1}, qx_{up2}, qa_{up3} \leftarrow$ **unpack64**(qa_p)
9:       $a_0 \leftarrow$ **vcvtepi32_ps**(qa_{up0})
10:       ... ▷ Repeat for the next 48 elements
11:       $o_0 \leftarrow$ **vfmadd**$(w_mul_qa_scale_vec, a_0, o_0)$
12:       ... ▷ Repeat for the next 48 elements
13:       **vstore**(O, o_0)
14:       ... ▷ Repeat for the next 48 elements
15:     **end for**
16:   **end for**
17: **end for**

3.2.4 Linear Backward Propagation

During the backpropagation of fully connected layers, we need to compute two gradients: $\partial L/\partial A$ and $\partial L/\partial W$ where L is the loss, A is the activation and W is the weight.

Computation of these gradients is shown through the equations 3.9:

$$\frac{\partial L}{\partial A} = s_o.s_w \frac{\partial L}{\partial O} Q_W^T, \\ \frac{\partial L}{\partial W} = s_a.s_o Q_A^T \frac{\partial L}{\partial O}. \tag{3.9}$$

All of the tensors in the right-hand side are quantized inputs and s_a, s_o, s_w are their corresponding scaling factors. $\frac{\partial L}{\partial O}$ is the quantized output gradient from the next layer. As the weights are sparse, the first equation represents a Sparse Matrix-Matrix multiplication, whereas the second one represents a Sampled Dense-Dense Matrix Multiplication. Similar to the forward algorithm, we use the transpose of the input tensors Q_A & $\frac{\partial L}{\partial O}$ to have streaming access over the activation, gradient output, and gradient input tensors.

Algorithm 10 Quantized Sparse Backward Algorithm for Linear Layers

1: **for** $i \leftarrow 0$ **to** N **with** i++ **do**
2: **for** $j \leftarrow W_idx_N[i]$ **to** W_idx_N[i + 1] **with** j++ **do**
3: $w_mul_O_scale_vec \leftarrow$ **vbroadcast**$(W_{val}[j] * s_w * s_o)$
4: $a_mul_O_scale_vec \leftarrow$ **vbroadcast**$(s_a * s_o)$
5: $acc_v \leftarrow$ **vbroadcast**(0)
6: **for** $b \leftarrow 0$ **to** B - 63 **with** b += 64 **do**
7: $da_0 \leftarrow$ **vload**$(dLdA + W_idx_M[j] * B + b)$
8: ... ▷ Repeat for the next 48 elements
9: $qa_{p0} \leftarrow$ **vload**$(Q_A + W_idx_M[j] * B + b)$
10: $qdo_{p0} \leftarrow$ **vload**$(dLdO + i * B + b)$
11: $qdo_{up0}, qdo_{up1}, qdo_{up2}, qdo_{up3} \leftarrow$ **unpack64**(qdo_{p0})
12: $do_0 \leftarrow$ **vcvtepi32_ps**(qdo_{up0})
13: ... ▷ Repeat for the next 48 elements
14: $s_0 \leftarrow$ **vfmadd**$(w_mul_O_scale_vec, do_0, da_0)$
15: ... ▷ Repeat for the next 48 elements
16: $acc_v \leftarrow$ **vdpbusdepi32**$(acc_v, qa_{p0}, qdo_{p0})$
17: **vstore**$(dLdA + W_idx_M[j] * B + b, s_0)$
18: ... ▷ Repeat for the next 48 elements
19: **end for**
20: $dLdW_val[j] \leftarrow$ **vhadd**(**vmul**$(a_mul_O_scale_vec,$ **vcvtepi32_ps**$(acc_v)))$
21: **end for**
22: **end for**

In the inner loop, we use the AVX512 VNNI instruction set to efficiently compute weight gradient updates using quantized, packed integer values of the activations and the output gradients. The **vdpbusdepi32**(src, a, b) instruction computes the value

$$\begin{aligned} dst[i] = src[i] &+ a.\text{byte}[4*i] * b.\text{byte}[4*i] \\ &+ a.\text{byte}[4*i+1] * b.\text{byte}[4*i+1] \\ &+ a.\text{byte}[4*i+2] * b.\text{byte}[4*i+2] \\ &+ a.\text{byte}[4*i+3] * b.\text{byte}[4*i+3]. \end{aligned} \tag{3.10}$$

While it is efficient to use this instruction as it operates on 8-bit integer values, it is not very straightforward to use because of its limitations. The instruction assumes a is an unsigned vector. Therefore, we need to do some preprocessing in algorithm 10 before we can use it. First, we extract the signs of the elements of both a and b. Then, we compute the signs of the elementwise multiplication, take the absolute value of a and elementwise

Convolution Dimension Definitions	
IC	Input Channel Count
OC	Output Channel Count
K	Kernel Size
M	Input Width
N	Input Height
OM	Output Width
ON	Output Height
B	Batch Size

Table 3.1: Symbols used to denote the dimensions of tensors in convolutional layers.

multiply the extracted signs with b. Alternatively, we could have used multiple fused multiply-add instructions instead and reduced them to a single vector. However, combined with unpacking and converting the quantized values into floats, using fused multiply-add operations become more expensive.

3.2.5 Convolutional Forward Propagation

For the forward algorithm of sparse convolutional layers, we implement a Sparse Tensor-Tensor convolution operation. Table 3.1 defines the symbols used to define the dimensions of tensors in the convolutional layers.

Input tensor A has dimensions $B \times IC \times M \times N$ and the weight W has the dimensions $OC \times IC \times K \times K$. Output O of convolution has dimensions $B \times OC \times OM \times ON$. Output dimensions OM and ON can be written as

$$\begin{aligned} OM &= \left\lceil \frac{M + 2 * padding - K + 1}{stride} \right\rceil, \\ ON &= \left\lceil \frac{N + 2 * padding - K + 1}{stride} \right\rceil. \end{aligned} \tag{3.11}$$

Here, padding denotes the size of the zero pads around the input and stride denotes the number of rows and columns skipped per slide.

For all $b \in [0, B)$, $oc \in [0, OC)$, $p \in [0, OM)$ & $q \in [0, ON)$ the output tensor O becomes

$$\begin{aligned} O[b, oc, p, q] = \sum_{ic=0}^{IC-1} \sum_{i=0}^{K-1} \sum_{j=0}^{K-1} & s_W . s_A . Q_W[oc, ic, i, j] \\ & . Q_A[b, ic, p + i * stride, q + j * stride]. \end{aligned} \tag{3.12}$$

Algorithm 11 shows our implementation of quantized, sparse forward pass for convolutional layers. Since weights are sparse, we can skip computations where $Q_W[oc, ic, i, j]$ are pruned which is what we do in our sparse convolution implementation. In our library we only support convolutional layers where stride is equal to either 1 or 2. For simplicity, we have shown the algorithm where stride is equal to 1. However, we can easily extend this algorithm to stride 2 by properly computing the bounds for p and q dimensions for O & Q_A tensors and incrementing the p and q indices of Q_A by 2 instead of 1.

The forward algorithm for convolutional layers is similar to the forward algorithm for fully connected layers introduced in subsection 3.2.3. To enable the use of SIMD instructions, again we placed the batch dimension B in the last position. However, as inputs and weights

for convolutional layers can't be transposed as in fully connected layers, we permute the input tensors such that their shapes become (IC, M, N, B).

Algorithm 11 Quantized Sparse Forward Algorithm for Conv. Layers

```
1: for oc ← 0 to OC with oc++ do
2:     for ic ← 0 to IC with ic++ do
3:         ic_start ← W_idx_OC[oc] + W_idx_IC[(IC + 1) * oc + ic]
4:         ic_end ← W_idx_OC[oc] + W_idx_IC[(IC + 1) * oc + ic + 1]
5:         for si ← ic_start to ic_end with si++ do
6:             w_mul_a_scale_vec ← vbroadcast(W_val[si] * s_a * s_w)
7:             p_s ← max(padding − W_x[si], 0)
8:             p_e ← min(padding − W_x[si] + M, OM)
9:             q_s ← max(padding − W_y[si], 0)
10:            q_e ← min(padding − W_y[si] + N, ON)
11:            for p ← p_s to p_e with p++ do
12:                for q ← q_s to q_e with q++ do
13:                    for b ← 0 to B - 63 with b+=64 do
14:                        o_0 ← vload(O)_{b,oc,p,q}
15:                        ...                              ▷ Repeat for the next 48 elements
16:                        qa_p ← vload(Q_A)_{b,ic,p,q}
17:                        qa_up0, qa_up1, qa_up2, qa_up3 ← unpack64(qa_p)
18:                        a_0 ← cvtepi32_ps(qa_up0)
19:                        ...                              ▷ Repeat for the next 48 elements
20:                        s_0 ← vfmadd(w_mul_a_scale_vec, a_0)
21:                        vstore(O, s_0)_{b,oc,p,q}
22:                        ...                              ▷ Repeat for the next 48 elements
23:                    end for
24:                end for
25:            end for
26:        end for
27:    end for
28: end for
```

3.2.6 Convolutional Backward Propagation

During the backward pass of dense convolutional layers, we compute the quantized dense partial gradients $\partial L/\partial A$ and $\partial L/\partial W$ as shown in equation 3.13. For simplicity, we only consider the case where padding equals 0 but in our implementation we generalize the equations to any padding.

The equations are valid for the following constraints: $\forall b \in [0, B)$, $\forall oc \in [0, OC)$, $\forall ic \in [0, IC)$, $\forall m \in [0, M)$, $\forall n \in [0, N)$, $\forall i \in [0, K)$ & $\forall j \in [0, K)$

$$
\begin{aligned}
\frac{\partial L}{\partial A}[b, ic, m, n] &= \sum_{oc=0}^{OC-1} \sum_{p=p_s}^{m} \sum_{q=q_s}^{n} s_O . s_W \frac{\partial L}{\partial O}[b, oc, p, q] \\
&\quad .Q_W[oc, ic, m-p, n-q], \\
\frac{\partial L}{\partial W}[oc, ic, i, j] &= \sum_{b=0}^{B-1} \sum_{p=0}^{M-K} \sum_{q=0}^{N-K} s_O . s_A \frac{\partial L}{\partial O}[b, oc, p, q] \\
&\quad .Q_A[b, ic, p + i * stride, q + j * stride].
\end{aligned}
\tag{3.13}
$$

Similar to forward pass, since weight Q_W is sparse, we can skip the computations of gradients where $Q_W[oc, ic, m-p, n-q]$ and $Q_W[oc, ic, i, j]$ are pruned for $\partial L/\partial A$ and $\partial L/\partial W$ respectively.

Algorithm 12 shows our implementation of the quantized sparse backward pass for convolutional layers. The loop structure is similar to Algorithm 11 whereas the actual computations are almost identical to Algorithm 10.

Algorithm 12 Quantized Sparse Backward Algorithm for Conv. Layers

```
1:  for ic ← 0 to IC with ic++ do
2:      for oc ← 0 to OC with oc++ do
3:          ic_start ← W_idx_OC[oc] + W_idx_IC[(IC + 1) * oc + ic]
4:          ic_end ← W_idx_OC[oc] + W_idx_IC[(IC + 1) * oc + ic + 1]
5:          for si ← ic_start to ic_end with si++ do
6:              w_mul_O_scale_vec ← vbroadcast(W_val[si] * s_W * s_O)
7:              a_mul_O_scale_vec ← vbroadcast(s_A * s_O))
8:              acc_v ← vbroadcast(0)
9:              p_s ← max(padding − W_x[si], 0)
10:             p_e ← min(padding − W_x[si] + M, OM)
11:             q_s ← max(padding − W_y[si], 0)
12:             q_e ← min(padding − W_y[si] + N, ON)
13:             for p ← p_s to p_e with p++ do
14:                 for q ← q_s to q_e with q++ do
15:                     da_p ← vload(dLdA)_{ic,p,q,k}
16:                     ...                              ▷ Repeat for the next 48 elements
17:                     qdo_p ← vload(dLdO)_{oc,p,q,k}
18:                     qa_p ← vload(QA)_{ic,p,q,k}
19:                     qdo_up0, qdo_up1, qdo_up2, qdo_up3 ← unpack64(qdo_p)
20:                     do_0 ← vcvtepi32_ps(qdo_up0)
21:                     ...                              ▷ Repeat for the next 48 elements
22:                     s_0 ← vfmadd(w_mul_O_scale_vec, do_0, da_p)
23:                     ...                              ▷ Repeat for the next 48 elements
24:                     acc_v ← vdpbusdepi32(acc_v, qa_p, qdo_p)
25:                     vstore(dLdA, s_0)_{ic,p,q,k}
26:                 end for
27:             end for
28:             dLdW_val[j] ← vhadd(vmul(a_mul_O_scale_vec, vcvtepi32_ps(acc_v)))
29:         end for
30:     end for
31: end for
```

For better performance, we again permute the batch layer to be in the final index and use AVX512 VNNI instructions to efficiently compute weight gradients. The preprocessing required to correctly compute accumulated $dLdW_val[j]$ value is identical to the preprocessing in subsection 3.2.4.

3.3 Multi-threaded Implementation

In sections 3.1 and 3.2 we have discussed the SIMD implementations of quantization functions as well as forward and backward algorithms for fully connected and convolutional layers. In order to further improve the performance we have implemented multi-threaded

versions of each kernel we have discussed so far. For completeness, in this chapter we will briefly discuss the multi-threaded implementations of our kernels.

3.3.1 Multi-threaded Quantization Kernels

Both the SAWB and Dithered quantization kernels have similar structure. In both of the kernels, we first compute the quantization scales according to algorithms 1 and 7 for SAWB and Dithered quantization respectively.

It is straightforward to see that both scale computation algorithms are embarrassingly parallel. We simply need to divide the loops across different threads and accumulate the results in different variables, one variable per thread.

To avoid repetition we will only show the parallel implementation of SAWB algorithms, however, it is straightforward to extend this to Dithered quantization algorithms as well.

Algorithm 13 displays the parallel version of algorithm 1. We have two new major steps. The first one is creating one accumulator per thread for each of the accumulator variables. The second step is reducing these accumulators into a single variable. The rest of the algorithm works exactly the same. The only caveat is, the scale computation loop is split amongst the available threads.

Algorithm 13 Multi-threaded Scale Computation for SAWB Quantization

1: $vacc_sq_mean \leftarrow$ **array**$[thread_count]$
2: $vacc_abs_mean \leftarrow$ **array**$[thread_count]$
3: **for** $i \leftarrow 0$ **to** $thread_count$ **with** $i++$ **do** ▷ Step 1
4: $\quad vacc_sq_mean[i] \leftarrow$ **vbroadcast**(0)
5: $\quad vacc_abs_mean[i] \leftarrow$ **vbroadcast**(0)
6: **end for**
7: **#parallel for**
8: **for** $i \leftarrow 0$ **to** $size(A) - 16$ **with** $i+ = 16$ **do**
9: $\quad tid \leftarrow$ **get_thread_id**$()$
10: $\quad val_A \leftarrow$ **vload**$(A)_i$
11: $\quad val_abs_A \leftarrow$ **vand**$(val_A,$ **vbroadcast**$(0x7FFFFFFFU))$
12: $\quad vacc_sq_mean[tid] \leftarrow$ **vfmadd**$(val_A, val_A, vacc_sq_mean[tid])$
13: $\quad vacc_abs_mean[tid] \leftarrow$ **vadd**$(val_abs_A, vacc_abs_mean[tid])$
14: **end for**
15: $acc_sq_mean_vec \leftarrow$ **vbroadcast**(0)
16: $acc_abs_mean_vec \leftarrow$ **vbroadcast**(0)
17: **for** $thread \leftarrow 0$ **to** $thread_count$ **with** $thread++$ **do** ▷ Step 2
18: $\quad acc_sq_mean_vec \leftarrow$ **vadd**$(acc_sq_mean_vec, vacc_sq_mean[thread])$
19: $\quad acc_abs_mean_vec \leftarrow$ **vadd**$(acc_abs_mean_vec, vacc_abs_mean[thread])$
20: **end for**

Multi-threaded implementation of quantization is even more straightforward than computing the scale. Quantization functions for both SAWB and Dithered quantization are also embarrassingly parallel. Splitting loop iterations between threads is enough to compute the quantized values efficiently.

The multi-threaded Dithered quantization algorithm slightly deviates from single-threaded implementation. Remember we mentioned that Dithered quantization is a stochastic quantization method. In order to generate random numbers for the noise, we initialized two AVX512 vectors of seeds and then used these seeds to create a sequence of vectorized random numbers. In order to avoid race conditions and contention on the seeds we need to

use a similar idea to what we have done in algorithm 13. Each thread needs its own pair of randomized AVX512 seed pairs. Algorithm 14 shows how we initialize the seeds using algorithm 6.

Algorithm 14 Multi-threaded Xorshift128+ Init Algorithm

1: $avx512_random_key1_perthread \leftarrow$ **array**$(thread_count)$
2: $avx512_random_key2_perthread \leftarrow$ **array**$(thread_count)$
3: **for** $thread \leftarrow 0$ **to** $thread_count$ **with** $thread$++ **do**
4: $avx512_random_key1_perthread[i], avx512_random_key2_perthread[i] \leftarrow$
5: **avx512_xorshift128plus_init**(
6: **get_random_uint64**(),
7: **get_random_uint64**()
8:)
9: **end for**

We then use these seeds to create unique sequences of random values in the multi-threaded version of algorithm 8. As quantization functions are also embarrassingly parallel, all of the algorithm except random number generation in dithered quantization stays the same. Algorithm 15 shows the modified parts of algorithm 8.

Algorithm 15 Multi-threaded Dithered Quantization

1: ...
2: **#parallel for**
3: **for** $i \leftarrow 0$ **to** $size(A) - 31$ **with** $i+ = 32$ **do**
4: ...
5: $tid \leftarrow$ **get_thread_id**()
6: $rnd_xor_0 \leftarrow$ **next_rnd**(▷ Compute random noise
7: $avx512_random_key1_perthread[tid]$,
8: $avx512_random_key2_perthread[tid]$
9:)
10: ...
11: **end for**
12: ...

3.3.2 Multi-threaded Propagation Kernels

As we have mentioned before, both the fully connected and convolutional layers have very similar loop structures because of the sparse representation of the weights. Multi-threaded implementations of the forward propagation for both of the layers are embarrassingly parallel. We simply need to split the outermost loops of algorithms 9 and 11 between the threads.

Multi-threaded implementations of the backward algorithms require a little more care, however. Remember, for both algorithms 10 and 12, computing the output gradients is a sparse tensor- tensor operation whereas computing the weight gradients is a sampled dense-dense tensor operation because of the sparsity of the weights. The same weight gradients can be updated through multiple loop iterations, creating a dependency between loops. Therefore, we need to have accumulators across threads and then reduce them to a single value.

In order to avoid being too repetitive, we will only show the pseudocode for multi-threaded convolutional backward propagation. However, the rest of the multi- threaded algorithms

are very similar. Algorithm 16 shows the modified parts of algorithm 12.

Algorithm 16 Multi-threaded Backward Algorithm for Conv. Layers

```
1:  #parallel for reduction(+:dLdW_val[nnz])                ▷ Elementwise reduction
2:  for ic ← 0 to IC with ic++ do
3:      for oc ← 0 to OC with oc++ do
4:          ...
5:          for si ← ic_start to ic_end with si++ do
6:              ...
7:              for p ← p_s to p_e with p++ do
8:                  for q ← q_s to q_e with q++ do
9:                      ...
10:                 end for
11:             end for
12:             dLdW_val[j] ← vhadd(...)                     ▷ One copy per thread
13:         end for
14:     end for
15: end for
```

Each thread has its own copy of sparse weights *dLdW_val* which are reduced into a single array when we exit the parallel section. The reduction function is summation.

Chapter 4

Evaluation

In this chapter, we evaluate the performance of the kernels described in chapter 3. We evaluate our kernels on three axes; finetuning accuracy, speed, and memory. In our evaluation, we compare our kernels against both the PyTorch and SparseProp [47] implementations. Before we dive into our evaluation, we introduce the execution environment we used to finetune models and collect experiment measurements.

The correctness of our implementations has been tested by comparing the outputs of quantized, sparse kernels against the outputs of dense linear and convolutional modules in PyTorch. To compute quantization error, we compute the mean-squared error (MSE) with respect to dense layer outputs and divide MSE by the square of two-norm of the dense outputs:

$$error = \frac{\frac{1}{N}\sum_{i=1}^{N}(out_{qi} - target_i)^2}{||target||_2^2} \tag{4.1}$$

Experimental Setup. We conducted our experiments using an Intel® Core^{TM} i7- 1068NG7 processor, which uses the Icelake architecture. It has a base frequency of 2.3 GHz. The cache sizes for L1, L2 and L3 caches are 80KiB (32 KiB **I** + 48 KiB **D**), 512KiB and 8MiB (shared), respectively. L1 and L2 caches are 8-way associative, and L3 cache is 16-way associative. The maximum memory bandwidth is 58.3GB/s [10]. We turn off the Intel Turbo Boost technology throughout the timing performance experiments. The C++ library is compiled using *clang* version 14.0.3. SIMD implementations of our kernels use AVX512 instructions. For multi-threaded implementations, we used OpenMP version 16.0.5. The compiler flags we used are: `-std=c++17 -march=icelake-client -mprefer-vector-width=512 -O3 -ffast-math -fno-finite-math-only -Xclang -fopenmp`.

The experiment code was written using Python version 3.9.17. To measure the execution time of forward and backward passes, we use the time module in Python. To track memory allocations, we use the PyTorch profiler [16] and for memory transfers, we simulate the data transfers from memory to the L3 cache using counters and hooks we implemented.

PyTorch Modules. Our quantized, sparse kernels are integrated into PyTorch modules to facilitate their use in our experiments and other applications. We convert our C++ library into a Python wheel using the Pybind11 library [28]. We use version 2.10.4 for our Pybind11 library.

For our end-to-end experiments, we use pre-trained and sparsified models in SparseZoo [15] accessible by sparseml library version 1.5.0.

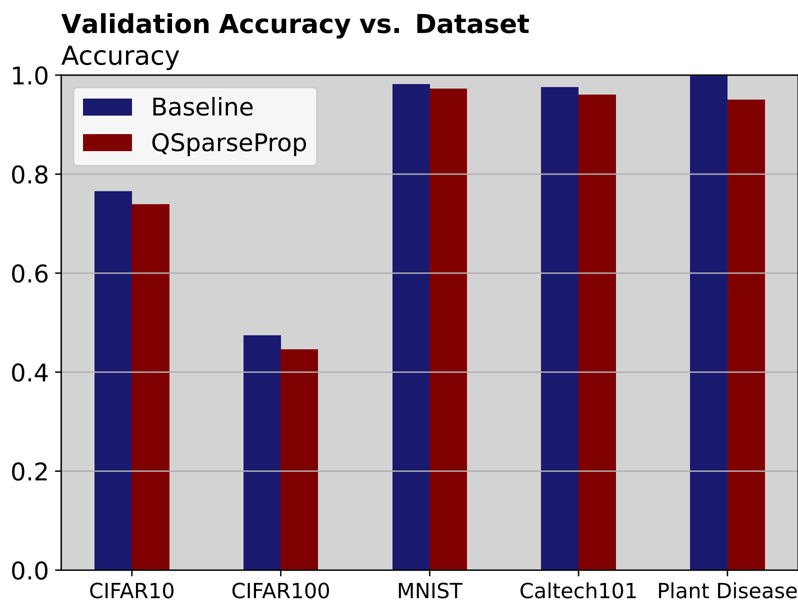


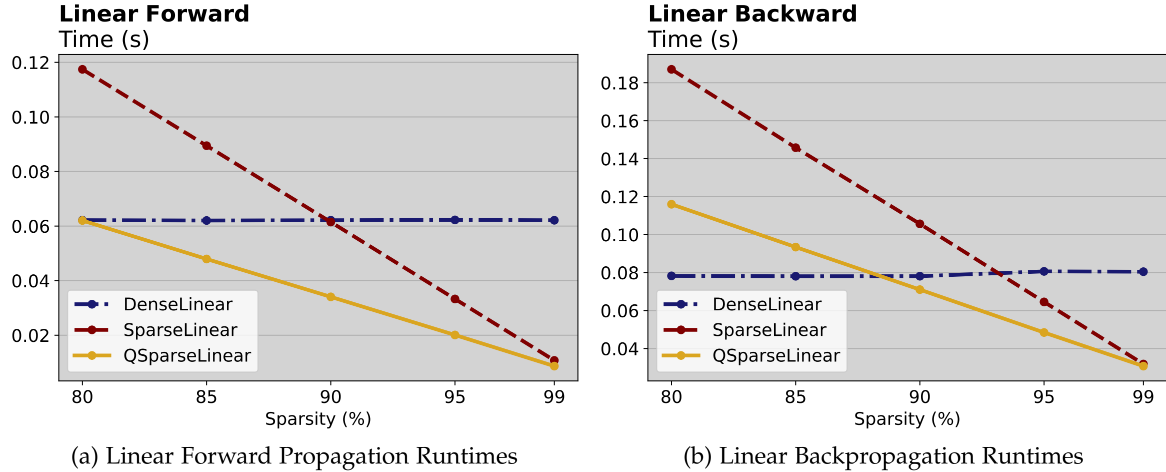
Figure 4.1: Top-1 validation accuracy of dense and quantized, 90% sparse ResNet50 models pre-trained on the ImageNet dataset.

4.1 Accuracy

As we have discussed previously, the quantization process introduces noise during dequantization and subsequent computations. On top of that, sparse tensor computations also introduce some noise. Therefore, in this section, we look into the effects of quantization on model finetuning accuracy for image classification tasks.

We analyze the accuracy drop during quantized finetuning for 90% uniformly pruned ResNet50 model. The model is pre-trained on ImageNet dataset. The model is pruned using AC/DC pruning [51]. The dense torchvision ResNet50 model trained on ImageNet has 76.15% Top-1 accuracy [17] whereas the dense pruned model has a Top-1 accuracy of 76.1% [13]. Figure 4.1 shows the finetuning performance of torchvision and quantized sparse model over five datasets. The models are finetuned for 15 epochs on CPU. The accuracy drop for the tested datasets is not significant. The average drop is 3.3% and the maximum drop is 6.1% for CIFAR100 dataset.

While the results in 4.1 are good, they do not paint the full picture. Inherent limitations of dithered quantization, the quantization scheme we use for gradient computations, have negative effects for more complex image classification tasks. As we have mentioned in subsection 3.1.3, Dithered quantization doesn't have a precise way of specifying the value range and controlling the bit counts in quantized values. Using a larger scaling factor reduces the bit count, however, in the process, it increases the sparsity of the quantized values as well. Increased sparsity has negative effects on accuracy. On the other hand, smaller scaling factors result in quantized values that are denser and use more bits. If the quantized values exceed 8-bit bounds, we have to clip them. Clipping these values introduces more bias as the count of quantized values with more than 8 bits increases. This prevents convergence in more complicated datasets we have tested like Caltech256 [22] and Stanford Cars [31].



(a) Linear Forward Propagation Runtimes

(b) Linear Backpropagation Runtimes

Figure 4.2: Runtimes of single-threaded forward and backward propagations of dense, sparse and quantized, sparse linear modules. Inputs have dimensions (B, M) = (902, 768). The module weights have dimensions (M, N) = (768, 3072)

4.2 Timing Performance

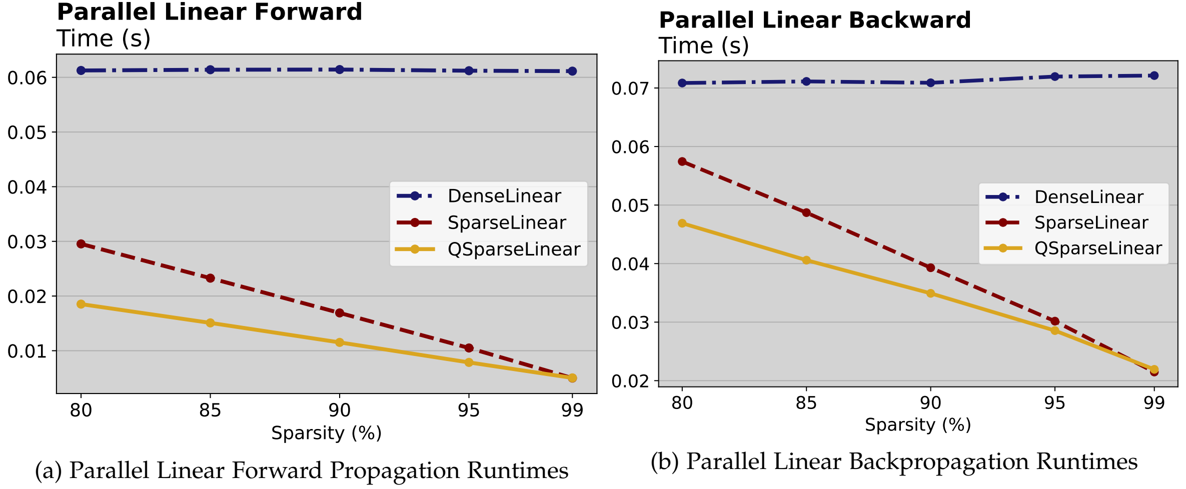
In this section, we analyze the speed improvement from using quantized, sparse modules against dense modules implemented in PyTorch and sparse modules implemented in SparseProp. We split the analysis of our quantized, sparse modules into two parts. In the first part, we analyze the time it takes to complete forward and backward passes for individual linear and convolutional modules. In the second part, we analyze the time it takes to complete the forward and backward passes during finetuning. In order to do that, we look at a case study of the ResNet50 model.

4.2.1 Module Timing Performance

In this subsection, we analyze the performance of quantized, sparse linear, and convolutional modules against their dense and sparse counterparts. In particular, we measure the runtime of forward and backward passes as a function of sparsity. We also look into the effects of multi-threading in the modules. For multi-threaded implementations, all modules use 8 threads.

Linear layers. We evaluate the performance of the linear layers by using input and weight tensors with dimensions (B, M) = (902, 768) and (M, N) = (768, 3072) respectively. For reference, the size of the tensors is larger than the capacity of the L3 cache of our system when the tensors are dense. In Figure 4.2, we show the runtimes of the single-threaded linear modules for forward and backward passes. For the forward pass, neither the sparse nor quantized, sparse module is faster than the dense module when the sparsity is at 80%. At 80% sparsity, the runtime of the forward pass for the quantized, sparse module is equal to the runtime of the dense module. The sparse module is around 2x slower. At 99% sparsity, the sparse and quantized, sparse modules are around 6x faster than the dense module.

For the backward pass, at 80% sparsity, the dense module is again faster than both of the sparse modules. The dense module is around 2.4x and 1.5x faster than the sparse and quantized, sparse modules when sparsity is 80%. Quantized, sparse module catches up to dense module at around 89% sparsity. Whereas the sparse module catches up to the dense module at around 94% sparsity. At 99% sparsity, both the sparse and quantized, sparse modules are around 2.6x faster than the dense modules. Remember that during training,



(a) Parallel Linear Forward Propagation Runtimes

(b) Parallel Linear Backpropagation Runtimes

Figure 4.3: Runtimes of multi-threaded forward and backward propagations of dense, sparse and quantized, sparse linear modules. Inputs have dimensions (B, M) = (902, 768). The module weights have dimensions (M, N) = (768, 3072). Thread number is 8.

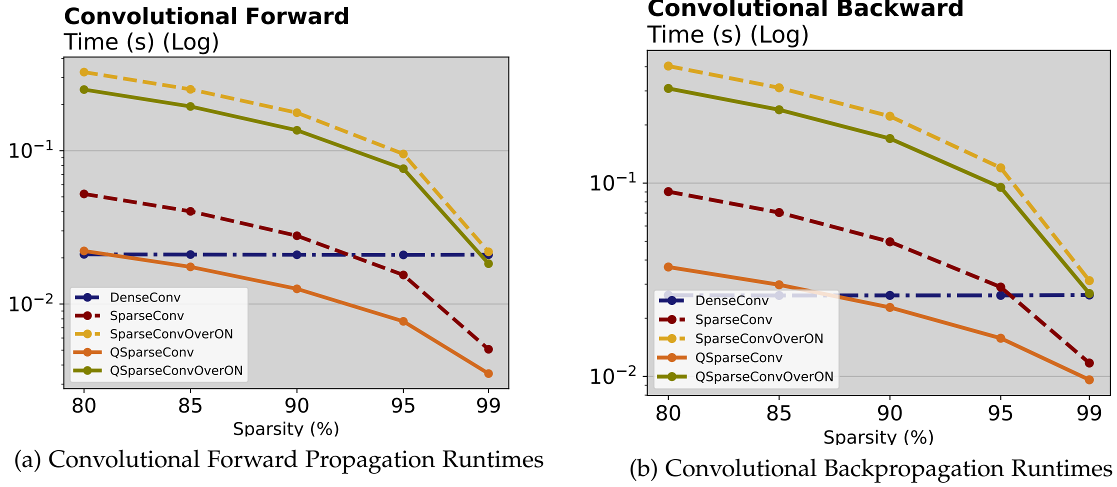
before the forward and backward propagations, we requantize the updated weights and quantize the new activations and gradients. The size of the activations and gradient tensors stay the same even when the sparsity of the weights increases. In single-threaded implementations, sparse modules are consistently slower than the quantized, sparse modules. As the sparsity of the weights increases, the cost of quantization of the activations, gradients, and weights increases relative to the cost of sparse forward and backward propagations. Therefore, the runtimes converge.

In Figure 4.3, we show the runtime comparison of the forward and backward passes for multi-threaded implementations of dense, sparse and quantized, sparse linear modules. The input and weight dimensions stay the same as in the single-threaded scenario. By the 80% sparsity mark, sparse and quantized, sparse modules are significantly faster than the dense module for both the forward and backward propagations. During the forward pass, at 80% sparsity, the sparse module is around 2x faster than the dense module. For the quantized, sparse module the speedup is around 3x. At 99% sparsity, both sparse and quantized, sparse modules are 7.5x faster than the dense module.

For backward passes, the situation is similar. By the 80% sparsity mark, the sparse linear module is around 1.16x faster, and the quantized, sparse linear module is around 1.4x faster. At 99% sparsity, both modules are around 3.5x faster than the dense module.

As we mentioned in section 3.3, the quantization functions are embarrassingly parallel. Therefore, they are easy to parallelize and benefit greatly from multi-threading as it can be seen from the Figures 4.2 and 4.3. Similar to the single-threaded case, the runtimes of the sparse and quantized, sparse modules converge as sparsity increases.

Convolutional layers. Similar to linear layers, for convolutional layers we also analyze both the single-threaded and multi-threaded implementations. Unlike linear layers, we have two different versions of convolutional layers for sparse and quantized, sparse implementations. While we were describing the implementations of the convolutional kernels in subsections 3.2.5 and 3.2.6 we mentioned that the input tensors for the convolutional layers were permuted such that batch index B was in the last position. In order to analyze the effects of this decision, we measure the runtimes of sparse and quantized, sparse modules against their counterparts where the inputs are not permuted. For the sparse modules,



(a) Convolutional Forward Propagation Runtimes

(b) Convolutional Backpropagation Runtimes

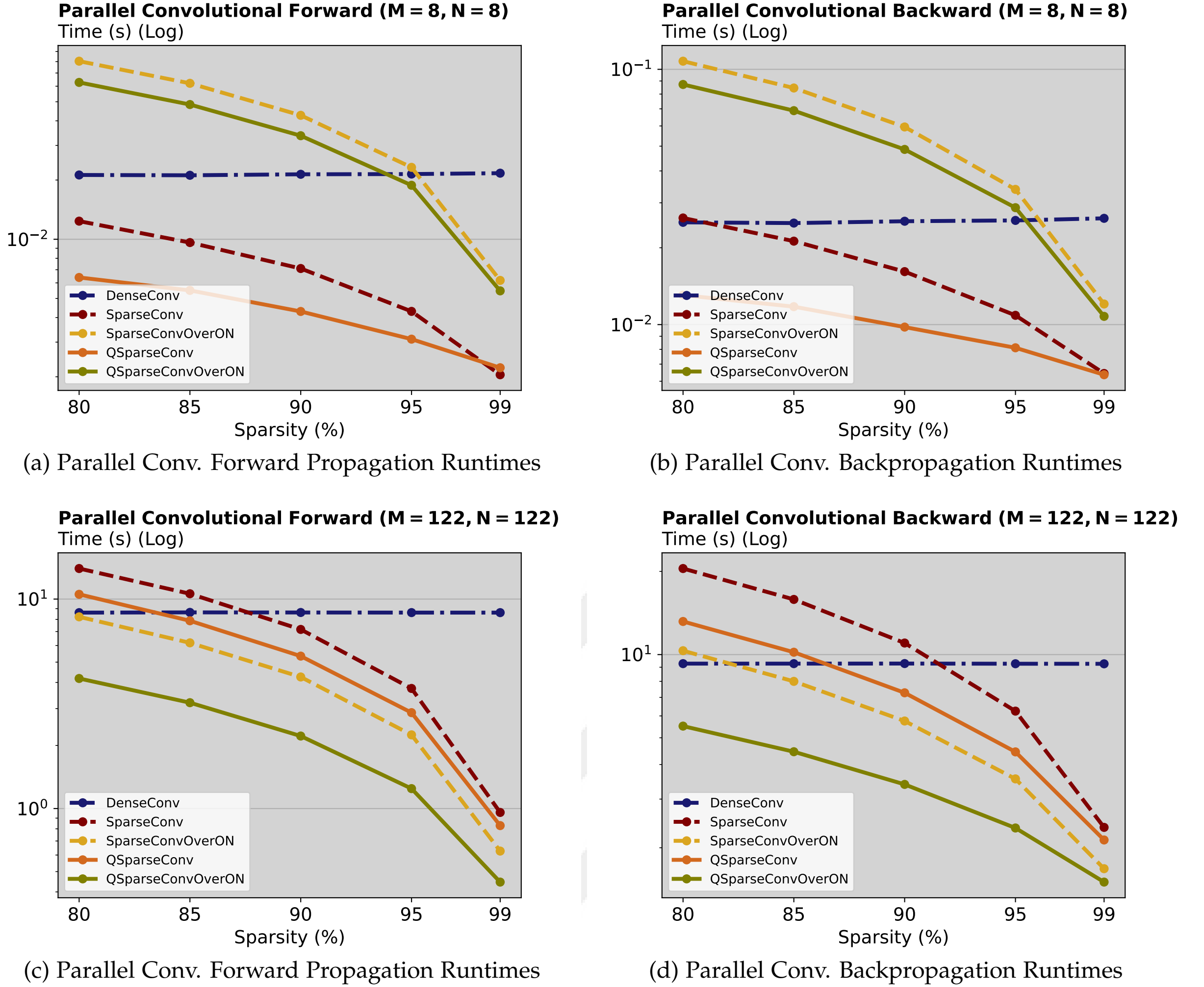
Figure 4.4: Runtimes of single-threaded forward and backward propagations of dense, sparse and quantized, sparse convolutional modules. Sparse and quantized, sparse implementations are compared against their counterparts vectorized over the ON dimension where the inputs are not permuted. Inputs have dimensions (B, IC, M, N) = (64, 128, 8, 8). The module weights have dimensions (IC, OC, K, K) = (128, 256, 3, 3). Stride is 1. Padding is 0.

we call these counterparts SparseConvOverON and for the quantized, sparse modules, QSparseConvOverON. OverON suffix signifies that the non-permuted versions are vectorized over the dimension ON in the innermost loop. To emphasize the speed differences between implementations, we use log plots.

For the single-threaded scenario, the input tensors have dimensions (B, IC, M, N) = (64, 128, 8, 8) and the dense weights have dimensions (IC, OC, K, K) = (128, 256, 3, 3). The forward and backward timing results draw similar pictures to their linear counterparts. At 80% sparsity, the dense convolutional forward operation is around 2.5x faster than the sparse module's forward pass. Quantized, sparse forward pass has the same runtime as its dense counterpart. Both modules vectorized over ON are significantly slower than the dense module. Dense module is around 15.5x faster than the sparse module vectorized over ON for the forward pass at 80% sparsity. This number is slightly better for the quantized, sparse module where the dense module is around 12x faster. The sparse module breaks even with the dense module at around 92% sparsity. Sparse and quantized, sparse modules vectorized over ON only break even when the sparsity is around 99%. At 99% sparsity, both the sparse and quantized, sparse modules are around 6.6x faster than the dense module.

For the backward pass, at 80% sparsity, no module is faster than the dense module. The dense module is 3.5x and 1.38x faster than the sparse and quantized, sparse modules respectively. For the modules vectorized over ON, the dense module is around 15.4x and 11.5x faster compared to the sparse and quantized, sparse modules respectively. The sparse module breaks even with the dense module at 95% sparsity whereas this number is around 87% for the quantized, sparse module. Modules vectorized over ON do not break even with the dense module until 99% sparsity. At 99% sparsity, speedups are 2.8x for sparse and quantized, sparse modules.

Similar to linear modules, quantized, sparse implementations are consistently faster than their sparse counterparts when there is only a single thread. Implementations vectorized over ON consistently perform poorer than their permuted counterparts. That is because when the M and N dimensions of the inputs (M = 8, N = 8) are small relative to vector sizes, it is more unlikely that the vectorized code sections will be executed due to a lack of contiguous non-zero elements. Therefore, permuting the batch index B into the last position and iterating over that allows us to use SIMD instructions more efficiently when



(a) Parallel Conv. Forward Propagation Runtimes

(b) Parallel Conv. Backpropagation Runtimes

(c) Parallel Conv. Forward Propagation Runtimes

(d) Parallel Conv. Backpropagation Runtimes

Figure 4.5: Runtimes of multi-threaded forward and backward propagations of dense, sparse and quantized, sparse convolutional modules. In figures 4.5a and 4.5b inputs have dimensions (B, IC, M, N) = (64, 128, 8, 8), and the module weights have dimensions (IC, OC, K, K) = (128, 256, 3, 3). In figures 4.5c and 4.5d inputs have dimensions (B, IC, M, N) = (64, 128, 122, 122) and the module weights have dimensions (IC, OC, K, K) = (128, 256, 3, 3). Stride is 1. Padding is 0.

B is large enough, i.e. 32, 64 etc.

In the multi-threaded setting, we consider two cases. In the first case, the input size is the same as in the single-threaded case where input tensors have dimensions (B, IC, M, N) = (64, 128, 8, 8). In the second case, the input tensors have dimensions (B, IC, M, N) = (64, 128, 122, 122). Weight tensors have the same dimensions as in the single-threaded scenario where the dimensions are (IC, OC, K, K) = (128, 256, 3, 3) for both cases. We use larger input tensors to showcase the runtime performance of kernels vectorized over ON. Even though we only use the permuted kernels in our quantized implementation, sparse and quantized, sparse implementations vectorized over ON have performance advantages for large M and N dimensions.

Figures 4.5a and 4.5b show the multi-threaded runtimes of the convolutional kernels when the input dimensions are (B, IC, M, N) = (64, 128, 8, 8). The results are quite similar to the single-threaded runtime results in Figures 4.4a and 4.4b as expected. In both the forward and backward propagations, the quantized, sparse modules are consistently faster than their sparse counterparts for all sparsities tested. Table 4.1 shows the speedups achieved against the dense implementations from the sparse and quantized, sparse implementations

Module \ Sparsity	Forward Speedups 80%	85%	90%	95%	99%	Backward Speedups 80%	85%	90%	95%	99%
Sparse	1.75x	2.3x	3x	5.25x	10.5x	0.96x	1.2x	1.56x	2.5x	4.15x
Q. Sparse	3.5x	4.2x	5.25x	7x	10.5x	1.9x	2.3x	2.8x	3.1x	4.2x
Sparse over ON	0.26x	0.34x	0.5x	0.91x	3.5x	0.25x	0.3x	0.42x	0.76x	2.1x
Q. Sparse over ON	0.33x	0.43x	0.64x	1.16x	4.2x	0.29x	0.37x	0.52x	0.89x	2.5x

Table 4.1: Speedups for sparse and quantized, sparse convolutional modules with respect to dense PyTorch modules with input tensor dimensions (B, IC, M, N) = (64, 128, 8, 8)

Module \ Sparsity	Forward Speedups 80%	85%	90%	95%	99%	Backward Speedups 80%	85%	90%	95%	99%
Sparse	0.61x	0.81x	1.2x	2.3x	8.95x	0.45x	0.59x	0.84x	1.49x	3.91x
Q. Sparse	0.81x	1.1x	1.6x	3x	10.4x	0.7x	0.9x	1.28x	2.1x	4.4x
Sparse over ON	1.05x	1.39x	2.02x	3.82x	13.7x	0.9x	1.16x	1.61x	2.62x	5.52x
Q. Sparse over ON	2.1x	2.7x	3.9x	6.9x	19.5x	1.7x	2.1x	2.73x	3.93x	6.18x

Table 4.2: Speedups for sparse and quantized, sparse convolutional modules with respect to dense PyTorch modules with input tensor dimensions (B, IC, M, N) = (64, 128, 122, 122)

for various sparsities. Both the sparse and quantized, sparse modules are faster than the dense module by 80% sparsity mark for both propagations. Even though quantized, sparse implementation is faster than the sparse implementation at lower sparsities, as sparsity increases the benefits of quantization diminish, and quantization overhead increases. Thus, at 99% sparsity, the sparse module catches up to the quantized, sparse module for both the forward and backward propagation. The modules vectorized over ON fail to utilize vectorization effectively, therefore, they don't even match dense PyTorch module until 95%+ sparsity.

Figures 4.5c and 4.5d plot the runtime of the dense, sparse, and quantized, sparse modules where the input tensor dimensions are (B, IC, M, N) = (64, 128, 122, 122). The figures show that convolutional kernels vectorized over ON become considerably faster when the M and N dimensions become larger. Table 4.2 shows the speedups of the sparse and quantized, sparse modules. We can see that modules vectorized over ON are almost twice as fast compared to their permuted counterparts when the M and N dimensions grow. Increasing these two dimensions allows us to use vectorized instructions better while avoiding permutation overheads, leading to significant performance gains. At 99% sparsity, quantized, sparse implementation vectorized over ON is around 19x and 6x faster than the dense module for forward and backward propagations respectively. Compared to the sparse implementation vectorized over ON, the quantized counterpart is around 1.35x and 1.12x faster for the forward and backward propagations respectively.

4.2.2 End-to-end Timing Performance

In the previous subsection, we focused on the timing performance of individual modules. While the analysis gives us useful insights with regards to the speedup of quantized, sparse module with respect to unquantized dense and sparse modules, we will not use these modules in isolation. In this subsection, we analyze the speedup achieved by the quantized, sparse kernels in the overall finetuning process. Similar to section 4.1, in order to perform our analysis, we will focus on the case study of finetuning a ResNet50 model using the CIFAR-10 dataset [33]. Sparse models are 90% uniformly pruned using AC/DC pruning

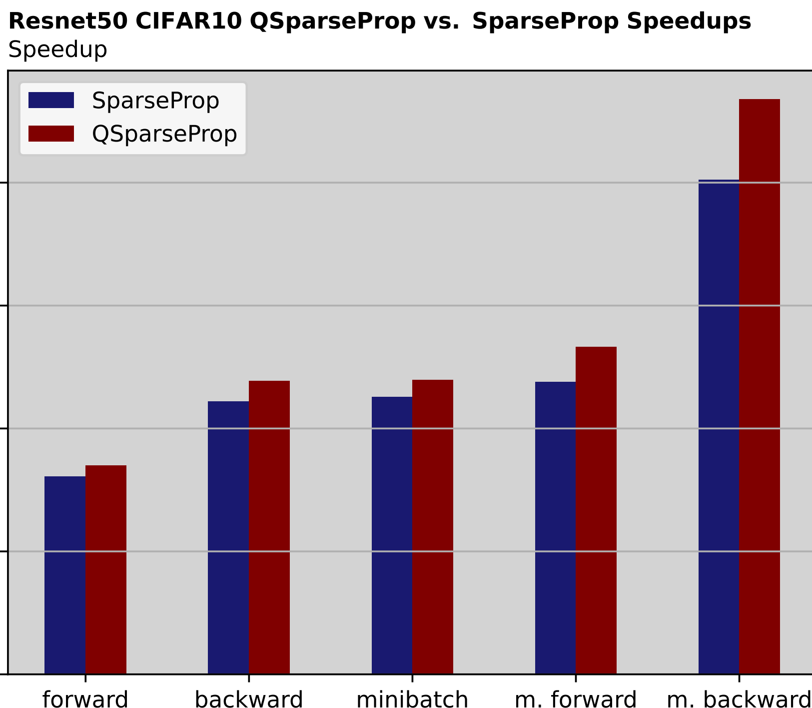


Figure 4.6: Finetuning speedup comparison of sparse and quantized, sparse ResNet50 with respect to the dense PyTorch model. We consider the speedups achieved during the end-to-end forward and backward passes, the speedup achieved in processing a minibatch of 64 CIFAR-10 images, and the speedups of forward and backward passes where we only consider the time spent in the convolutional and linear layers.

[51].

During our end-to-end timing analysis, we focus on five aspects. Figure 4.6 shows the speedups achieved by the sparse and quantized, sparse ResNet50 during the finetuning process with respect to the dense PyTorch model. We analyze the speedups of end-to-end forward and backward passes, the speedup achieved during the processing of a minibatch of size 64, and the speedup of only convolutional and linear layers during the forward and backward passes. The end-to-end speedup for forward pass is 1.6x and 1.7x for sparse and quantized, sparse models. Backward passes achieve slightly better speedups of 2.2x for sparse models and 2.4x for quantized, sparse models. Minibatch speedups are similar to end-to-end backward pass with 2.25x for sparse and 2.4x for quantized, sparse models. For the speedups in only convolutional and linear layers, the sparse model achieves 2.37x speedup whereas the quantized, sparse model achieves 2.66x speedup. For backward pass, this number is around 4x and 4.67x for sparse and quantized, sparse models respectively. For both the sparse and quantized, sparse model finetuning, the speedups in convolutional and linear modules are significantly higher than the overall speedups in end-to-end forward and backward propagations and minibatch processing which indicates costly overheads outside of the convolutional and linear layers.

In end-to-end setting quantized, sparse model achieves around 1.12x and 1.16x speedup during the forward and backward passes respectively against the sparse model. This speedup is more moderate compared to the speedup 90% sparse, quantized module achieved against the 90% sparse module in subsection 4.2.1. The most likely reason for this modest speedup is because the weight tensors are smaller in ResNet50 compared to the weights we used in the previous subsection. This means that the benefits of quantization are less pronounced because the weights are smaller. We can also see this effect in the previous sub-

section where the speedup of quantized, sparse modules with respect to sparse modules decreases as the sparsity of the modules increases.

4.3 Memory Usage

As we mentioned previously, the primary objective of quantization is to decrease the data transfers in memory bottlenecked programs to achieve performance increases. Therefore, in this section of our evaluation, we turn our focus to memory operations performed in our library. We split our analysis into two parts. The first one is analyzing the memory allocated during the forward and backward passes of the quantized, sparse modules against dense and sparse modules. Secondly, we analyze the data transferred from memory to the L3 cache during the finetuning process. For these experiments, the modules are multi-threaded and use 8 threads.

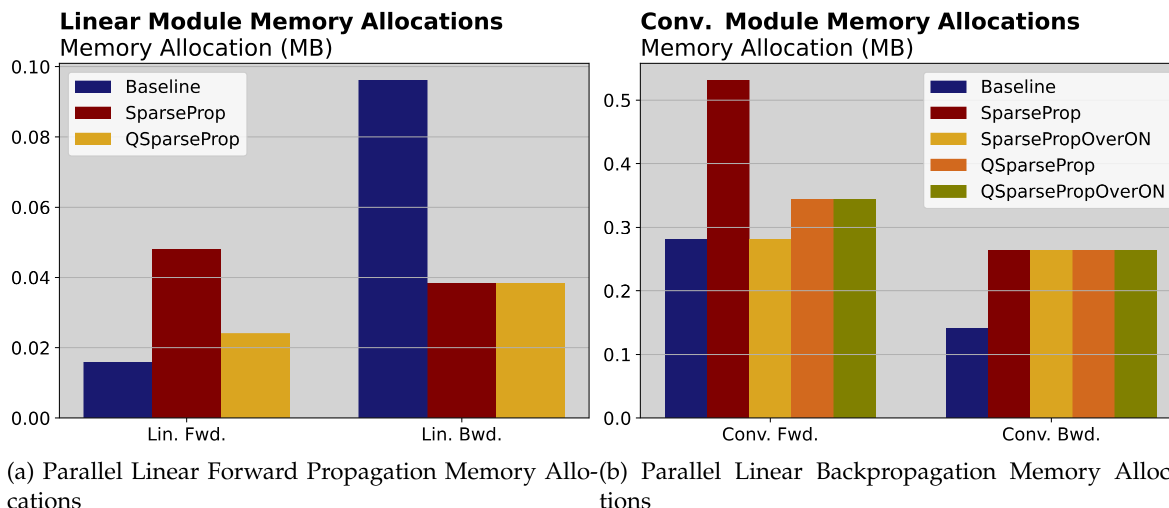
4.3.1 Memory Allocation

In this subsection, we analyze the amount of memory allocated during the forward and backward propagations. We further split our analysis into two. First, we measure the memory allocated in individual linear and convolutional layers. Then we measure the end-to-end memory allocation in the ResNet50 model. As we mentioned previously in the experiment setup at the beginning of the chapter, we use the PyTorch profiler to measure the memory allocation of the modules. It is important to notice that the PyTorch profiler measures the net memory allocation of code segments. This means that the memory allocation indicates the total memory that still remains allocated when we exit the scope of the measured code segment.

Module Memory Allocation Figure 4.7 shows the memory allocated during the forward and backward passes of linear and convolutional modules. We start our analysis by turning our attention to Figure 4.7a which shows the memory allocated during the forward and backward propagations of linear layers. The input tensors for linear modules have dimensions (B, M) = (128, 512). Weights of the linear modules have dimensions (N, M) = (256, 512). Sparse and quantized, sparse modules are both 90% sparse. The weights of the quantized module are 8-bit quantized.

During the forward pass of the linear modules, the dense module allocated 16KB, the sparse module allocated 48KB, and the quantized, sparse module allocated 24KB of space. Both the sparse and quantized, sparse modules require more space during the linear forward pass. We presume this is because, in order to decrease random memory accesses during the computations, we take the transpose of the input tensors. To avoid segmentation faults we need to convert transposed tensors into contiguous ones which requires extra space. We use these tensors in the proceeding computations as well. What is surprising, however, is the discrepancy between the memory allocations of sparse and quantized, sparse modules. We allocate extra space to quantize the inputs in the quantized modules but sparse modules still allocate twice as much space. A potential explanation is that we save the transposed input tensor for the backward pass which increases the net memory allocation. In the quantized module, this tensor occupies 75% less space compared to the sparse module due to quantization.

In the backward pass of linear modules, we have a different situation. The dense module allocates 96KB, the sparse and quantized, sparse modules allocate 38KB of space. The memory allocations in the sparse and quantized, sparse modules are identical, therefore, it is expected that the space allocated is equal. In the backward pass, we allocate space for the input and weight gradients. Even though, the sparse and quantized, sparse modules



(a) Parallel Linear Forward Propagation Memory Allocations

(b) Parallel Linear Backpropagation Memory Allocations

Figure 4.7: Memory allocations of linear and convolutional modules during the forward and backward passes. The inputs to linear modules have dimensions (B, M) = (128, 512). Linear weights have dimensions (N, M) = (256, 512). Input tensors to convolutional layers have dimensions (B, IC, M, N) = (64, 128, 8, 8). The convolutional weights have dimensions (OC, IC, K, K) = (256, 128, 3, 3). Stride is 1. Padding is 0. Sparse modules have 90% sparsity.

have the same transpose and contiguous space overheads, since the weight tensors are 90% sparse, they save considerable amounts of space for the weight gradients compared to the dense module.

Figure 4.7b shows the memory allocated during the forward and backward passes of convolutional layers. The input tensors of the convolutional layers have dimensions (B, IC, M, N) = (64, 128, 8, 8). The convolutional weights have dimensions (OC, IC, K, K) = (256, 128, 3, 3). During the forward pass, the dense module allocates 281KB, sparse and quantized, sparse modules allocate 531KB and 343KB respectively, sparse and quantized, sparse modules vectorized over ON allocate 281KB and 343KB respectively. The sparse module allocates the most amount of space because it permutes the input tensor and uses the permuted tensor for the subsequent computations. The quantized modules also allocate space for the permuted input tensors, however, as these modules also quantize the input tensor, the space allocation is around 64% of the sparse module. The sparse module vectorized over ON has the same space allocation as the dense module as it doesn't permute the input tensors and only allocates space for the output tensor. Since the output tensor is dense and unquantized, the net memory allocation in the sparse module vectorized over ON is exactly equal to the space allocation in the dense module.

For the backward pass, the dense module allocates 141KB, and the rest of the modules allocate 263KB. Similar to linear modules, it makes sense that the sparse modules allocate the same amount of space as they all need to allocate the same space for the dense input tensor gradients and the sparse weight gradients. What is confusing, however, is that the dense module allocates significantly less memory during the backpropagation. This is unexpected and in fact, contradictory to our measurements in the next segment. Weight gradients of the dense module should occupy 10x more space and therefore, should probably allocate more space during the backward pass.

End-to-end Memory Allocation Figure 4.8 shows the memory allocated during a single forward and backward pass of the ResNet50 model. The input to the model has dimensions (B, IC, M, N) = (64, 3, 32, 32). Similar to the previous segment, the sparse models have 90% sparsity. During the forward pass of convolutional layers, the dense model allocates

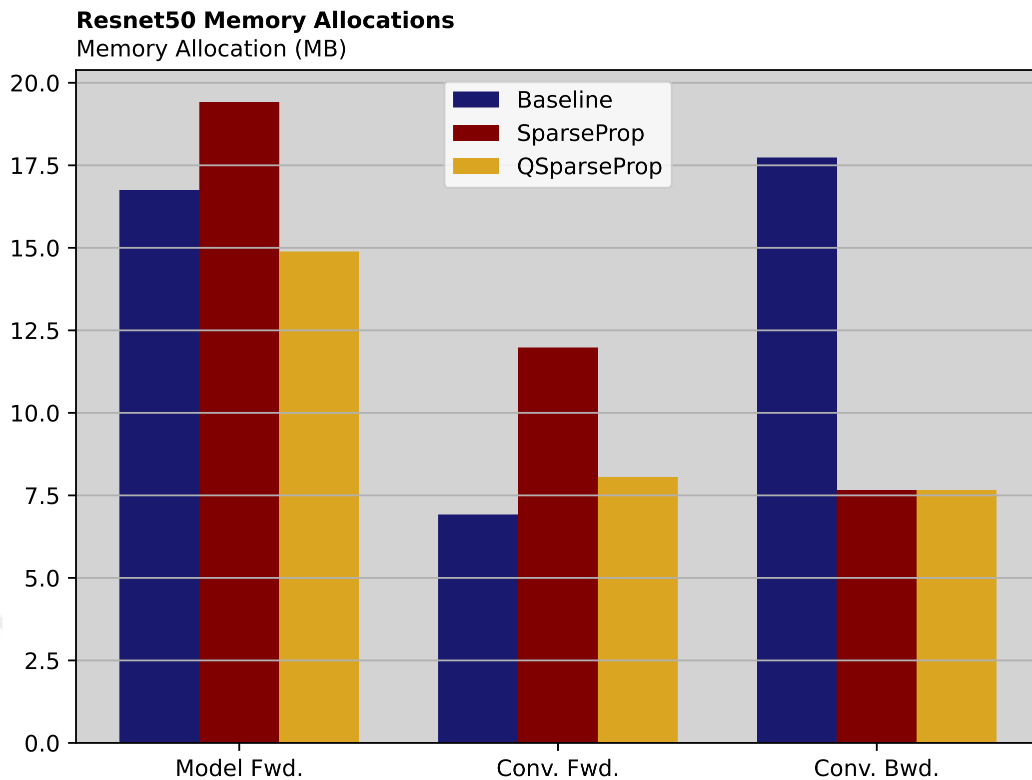


Figure 4.8: Memory allocated during the forward and backward passes of a ResNet50 model. The model has 10 output classes. Input tensors have dimensions (B, IC, M, N) = (64, 3, 32, 32). Sparse models are 90% sparse.

6.92MB, the sparse model allocates 12MB and the quantized, sparse model allocates 8MB of space. In the backward pass of the convolutional layers the dense, sparse, and quantized, sparse models allocate 17.73MB, 7.66MB, and 7.66MB respectively. These results look very similar to the memory allocation results of linear modules in Figure 4.7a instead of the results of the convolutional modules in Figure 4.7b. Sparse convolutional layers use around 0.4x of the space used in the dense layers during the backward pass, presumably because of the sparsity of the weights. This ratio (space allocated to sparse models vs space allocated to dense model) of space allocated is actually equal to the ratio of the space allocated in the linear modules in the previous segment.

During the end-to-end forward pass of the ResNet50 model, the dense model allocates 16.7MB, the sparse model allocates 19.4MB and the quantized, sparse model allocates 14.8MB of space. Even though the sparse and quantized, sparse models allocate more space during the forward pass of the convolutional layers, they manage to amortize the total space allocated in the end-to-end forward pass. The sparse model uses 1.16x (compared to the dense model) more space in the end-to-end forward pass as opposed to 1.73x space used in the convolutional layers. Similarly, compared to the dense model, the quantized, sparse model uses 0.88x space as opposed to the 1.15x space used in the convolutional layers.

As we have mentioned in the previous segment, the memory allocation of the convolutional layers during the backward pass is contradictory to the measurements we have in Figure 4.8. The measurements we have for this scenario make more sense, however, as we expect to allocate significantly less space in the sparse backward passes due to the sparsity of the weights. We find it surprising that the quantized model allocates less space in the end-to-end forward pass even though this is not the case in the forward pass of the convolutional layers. In the convolutional layers, sparse and quantized, sparse modules should allocate

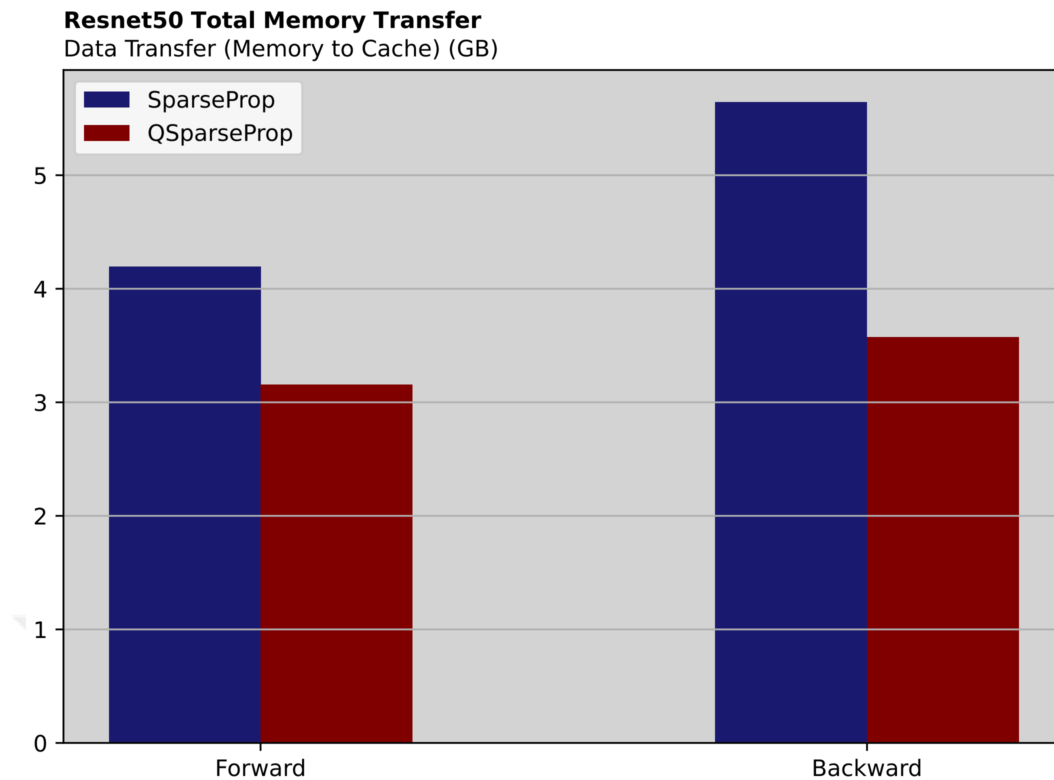


Figure 4.9: Total simulated data transfers from memory to L3 cache during the forward and backward passes of a ResNet50 model. The model has 10 output classes. The input tensors have dimensions (B, IC, M, N) = (64, 3, 32, 32).

more space due to permutation. The space allocation in the other hidden layers, however, should be equal to the space allocated in the dense module. Therefore, it is unexpected, and rather implausible, for the quantized, sparse model to allocate less space in the overall end-to- end forward pass compared to the dense module.

4.3.2 Data Transfers

In this subsection, we analyze the amount of data transferred from the main memory to L3 cache. In order to measure the data transfers from memory to cache, we simulate data transfers that take place in the forward and backward propagations during the finetuning of a quantized, 90% sparse ResNet50 model using our own Python hooks and counters. The model takes input tensors of dimensions (B, IC, M, N) = (64, 3, 32, 32) and has 10 output classes. In this subsection, to simplify the analysis, in our counters we assume that every memory access is a cache miss. While unrealistic, it is not totally impossible since the sparsity of the models is high and the size of the input and output tensors is large. Therefore, the results presented in this subsection are for the worst-case scenario and an overestimation of the actual amount of data transferred from memory to cache.

Figure 4.9 shows the simulated amount of data transferred from memory to the L3 cache during the convolutional forward and backward propagations. During the end-to-end forward pass, the convolutional layers of the sparse ResNet50 are projected to transfer a total of 4.2GB of data. In contrast, the quantized model transfers around 3.1GB of data, a reduction of 27%. The end-to-end backward pass of the sparse convolutional layer transfers around 5.6GB of data while the quantized model transfers around 3.6GB of data, a 36% reduction. During the forward passes, our library quantizes both the activations and the sparse weights. The convolutional outputs remain unquantized until the next layer. In the

backward pass, in addition to these tensors, we also quantize the output gradients while the sparse weight gradients and the input gradients remain unquantized.

Both models perform roughly the same amount of floating point operations during the forward and backward passes, 1.7 billion flops for the forward pass and 3.9 billion for the backward pass. Therefore, on a low estimate, convolutional layers of the quantized model perform around 35% more flops per byte transferred for the forward propagation while they perform around 55% more flops per byte transferred for the backward propagations. Based on our flop and memory transfer estimates, we expect both the sparse and quantized, sparse implementations to be still in the memory bound region. However, the operational intensity of the models suggests that both models are far away from the performance upper bound. This highlights the difficulty of optimizing operations with unstructured sparsity. Even though the memory transfer is reduced considerably, we are still far from effectively utilizing the underlying hardware.

Chapter 5

Conclusion

In this thesis, we introduced a high-performance library for quantized, sparse finetuning of neural networks. We built on top of an existing sparse training framework, SparseProp, to extend it with quantization support. We implemented parallel, SIMD versions of two quantization methods, SAWB and Dithered quantization, to quantize weights, activations, and gradients in neural networks. We implemented a novel INT8 version of Dithered quantization which is actually an FP8 quantization method. We integrated our high-performance kernels into linear and convolutional modules in PyTorch to facilitate their use during finetuning.

In the linear and convolutional layers of ResNet50, using our framework, 90% sparse, quantized models achieved around 2.66x speed up during the forward pass and 4.67x speed up during the backward pass compared to dense PyTorch models. Compared to models using SparseProp modules, we achieved around 1.1x and 1.2x speed up in the forward and backward passes respectively. We managed to maintain around 96.5% of the dense finetuning accuracy while reducing the total training time and memory allocation in convolutional layers by more than half. We reduced the total data transfers from memory to L3 cache by 30-35%. Even though we use 8-bit quantization methods on activations, weights, and gradients, we did not achieve around the 4x speed up as expected from 8-bit quantization because of the cost of requantizing the tensors.

Future Work While testing our framework, we only focused on image classification tasks using ResNet architecture. In the future, it is possible to extend the framework to other architectures like BERT and other tasks such as text classification.

Moreover, the effectiveness of our quantization framework suffered due to the inherent limitations of Dithered quantization. Due to its floating point quantization origins, dithered quantization does not compute a scale that allows it to adjust the number of bits in the quantized values. Extending the framework with an unbiased integer quantization method with adjustable bit width would be worthwhile.

Additionally, for convolutional layers different quantization granularities can be explored. In our early work, we looked into fixed size quantization blocks similar to the ones used in [58]. We abandoned that idea, as there wasn't a noticeable increase in the validation accuracy of the models we tested. However, one can look at quantization granularities like filter-wise quantization to reduce quantization error further.

Appendix A

Stochastic/Deterministic Quantization

The proofs in this section are taken from [7]. They are used to introduce stochastic quantization in section 2.2 and justify our choice of gradient quantization functions in subsection 3.1.3.

Theorem A.1 *Stochastic Rounding schemes are unbiased.*

Proof Assume we have a real number denoted by x. x is quantized into a bin with lower bound *l(x)* and upper bound *u(x)*. Then, stochastic rounding can be defined as:

$$SR(x) = \begin{cases} l(x) & \text{w.p. p(x) = 1 - } \frac{x-l(x)}{u(x)-l(x)} \\ u(x) & \text{w.p. p(x) = } \frac{x-l(x)}{u(x)-l(x)} \end{cases} \tag{A.1}$$

The expected rounding value can be written as:

$$E[SR(x)] = l(x).p(x) + u(x).(1-p(x)) = x \tag{A.2}$$

The expectation is over the randomness of SR. Then we can write bias as:

$$Bias[SR(x)] = E[SR(x) - x] = E[SR(x)] - x = 0 \tag{A.3}$$

Therefore, SR(x) is an unbiased rounding scheme. □

Theorem A.2 *Round-to-nearest (RDN) rounding scheme has lower mean squared error (MSE) than stochastic quantization.*

Proof The variance of SR can be written as:

$$\begin{aligned} Var[SR(x)] &= (l(x) - E[SR(x)]^2.p(x) \\ &+(u(x) - E[SR(x)])^2.(1-p(x)) \\ &= (x - l(x)).(u(x) - x) \end{aligned} \tag{A.4}$$

Last equality can be obtained by substituting SR[x] and p(x).

The bias of RDN can be written as:

$$Bias[RDN(x)] = min(x - l(x), u(x) - x) \tag{A.5}$$

The variance of RDN is equal to 0 as it is a deterministic scheme.

MSE for a rounding scheme R(x) can be written as:

$$MSE[R(x)] = E[R(x) - x]^2 = Var[R(x)] + Bias^2[R(x)] \tag{A.6}$$

Then, we get the following distortion:

$$MSE = \begin{cases} [min(x - l(x), u(x) - x)]^2 & \text{for RDN(x)} \\ (x - l(x)).(u(x) - x) & \text{for SR(x)} \end{cases} \tag{A.7}$$

Since $min(a,b)^2 \leq a.b$ then we have:

$$MSE(SR(x)) \geq MSE(RDN(x)) \tag{A.8}$$

□

Theorem A.3 *Stochastic gradient quantization produces unbiased approximation of unquantized gradients.*

Proof Denote the pre-activation and activation values of layer l of an L layer neural network as z_l and a_l respectively. C denotes the cost function of the neural network. δ_l denotes the derivative of the loss with respect to inputs and outputs. From the chain rule we have:

$$\delta_l \triangleq \frac{dC}{dz_l} = \frac{da_l}{dz_l} ... \frac{da_L}{dz_L}\frac{dC}{da_L} \tag{A.9}$$

Then, we can define the derivative of loss recursively $\forall l < L$

$$\delta_l \triangleq \frac{da_l}{dz_l}\frac{dz_{l+1}}{da_l}\delta_{l+1} \tag{A.10}$$

Denote quantization function as *Q(.)* and quantized derivatives of layer l as δ_l^q. Then we have

$$\delta_l^q \triangleq Q(\frac{da_l}{dz_l}\frac{dz_{l+1}}{da_l}\delta_{l+1}^q) \tag{A.11}$$

Let us assume *Q(.)* is a stochastic quantization function where $E[Q(x)] = x$. Then we have

$$\begin{aligned} E\delta_l^q &= EQ(\frac{da_l}{dz_l}\frac{dz_{l+1}}{da_l}\delta_{l+1}^q) \overset{(1)}{=} E[E[Q(\frac{da_l}{dz_l}\frac{dz_{l+1}}{da_l}\delta_{l+1}^q)|\delta_{l+1}^q]] \\ &\overset{(2)}{=} E[\frac{da_l}{dz_l}\frac{dz_{l+1}}{da_l}\delta_{l+1}^q] \overset{(3)}{=} \frac{da_l}{dz_l}\frac{dz_{l+1}}{da_l}E\delta_{l+1}^q = \frac{da_l}{dz_l}\frac{dz_{l+1}}{da_l}\delta_{l+1} \end{aligned} \tag{A.12}$$

Here we used the following properties:

1. Law of total expectation
2. E[Q(x)] = x
3. Linearity of backpropagation

Denote the gradient of weights in layer l as $\nabla_{W_l}C = \delta_l a_{l-1}^T$. Then the quantized gradient is denoted as $\nabla_{W_l}C_q = \delta_l^q a_{l-1}^T$. From equation A.12, we can prove that the quantized gradient is an unbiased estimator:

$$E[\nabla_{W_l}C_q] = E[\delta_l^q a_{l-1}^T] = E[\delta_l^q]a_{l-1}^T = \delta_l a_{l-1}^T = \nabla_{W_l}C \tag{A.13}$$

□

Appendix B

Packing/Unpacking Algorithms

Here, we share the pseudocode for the packing and unpacking algorithms introduced in subsection 3.1.1.

Algorithm 17 Pack32

Input: vec_{q0}, vec_{q1}
Output: $vpacked$

1: $vec_lo_{q0} \leftarrow$ **vcastsi512_si256**(vec_{q0}) ▷ Split 512-bit vectors into two 256-bit vectors
2: $vec_hi_{q0} \leftarrow$ **vextracti32x8_epi32**(vec_{q0})
3: $vec_lo_{q1} \leftarrow$ **vcastsi512_si256**(vec_{q1})
4: $vec_hi_{q1} \leftarrow$ **vextracti32x8_epi32**(vec_{q1})
5: $left_0 \leftarrow$ **vslli**$(vec_lo_{q0}, 24)$
6: $left_1 \leftarrow$ **vslli**$(vec_hi_{q0}, 24)$
7: $left_2 \leftarrow$ **vslli**$(vec_lo_{q1}, 24)$
8: $left_3 \leftarrow$ **vslli**$(vec_hi_{q1}, 24)$
9: $right_0 \leftarrow$ **vsrli**$(vec_lo_{q0}, 24)$
10: $right_1 \leftarrow$ **vsrli**$(vec_hi_{q0}, 16)$
11: $right_2 \leftarrow$ **vsrli**$(vec_lo_{q1}, 24)$
12: $right_3 \leftarrow$ **vsrli**$(vec_hi_{q1}, 16)$
13: $packed_{01} \leftarrow$ **vor**$(right_0, right_1)$ ▷ Combine shifted halves
14: $packed_{23} \leftarrow$ **vor**$(right_2, right_3)$
15: $interleave_lo \leftarrow$ **vpermute**$(packed_{01}, 0x20)$
16: $interleave_hi \leftarrow$ **vpermute**$(packed_{23}, 0x31)$
17: $shuffle_lo \leftarrow$ **vshuffle_epi8**$(interleave_{l}o, shuffle_lo_mask)$
18: $shuffle_hi \leftarrow$ **vshuffle_epi8**$(interleave_hi, shuffle_hi_mask)$
19: $vpacked \leftarrow$ **vor**$(shuffle_lo, shuffle_hi)$
20: **return** $vpacked$

In algorithm 17, $shuffle_lo_mask$ and $shuffle_hi_mask$ are special masks which are used to shuffle 8-bit integers into the right locations in the vectors. They are defined as

$$
\begin{aligned}
&shuffle_lo_mask = \textbf{vset_epi8}(\\
&\quad 0,4,8,12,2,6,10,14, \\
&\quad 1,5,9,13,3,7,11,15, \\
&\quad 0,4,8,12,2,6,10,14, \\
&\quad 1,5,9,13,3,7,11,15 \\
&), \\
&shuffle_hi_mask = \textbf{vset_epi8}(\\
&\quad 2,6,10,14,0,4,8,12, \\
&\quad 3,7,11,15,1,5,9,13, \\
&\quad 2,6,10,14,0,4,8,12, \\
&\quad 3,7,11,15,1,5,9,13 \\
&).
\end{aligned}
\tag{B.1}
$$

Algorithm 18 Unpack32

Input: *vpacked*

Output: $vec_{q0}, vec_{q1}, vec_{q2}, vec_{q3}$

1: $v_switched \leftarrow \textbf{vpermute}(vpacked, vpacked, 0x21)$
2: $v_half_0 \leftarrow \textbf{vshuffle_epi8}(vpacked, restore_lo_mask)$
3: $v_half_1 \leftarrow \textbf{vshuffle_epi8}(v_switched, restore_hi_mask)$
4: $v_full \leftarrow \textbf{vor}(v_half_0, v_half_1)$
5: $left_0 \leftarrow \textbf{vslli}(v_full, 24)$
6: $left_1 \leftarrow \textbf{vslli}(v_full, 16)$
7: $left_2 \leftarrow \textbf{vslli}(v_full, 8)$
8: $left_3 \leftarrow \textbf{vslli}(v_full, 0)$
9: $vec_{q0} \leftarrow \textbf{vsrai}(left_0, 24)$
10: $vec_{q1} \leftarrow \textbf{vsrai}(left_1, 24)$
11: $vec_{q2} \leftarrow \textbf{vsrai}(left_2, 24)$
12: $vec_{q3} \leftarrow \textbf{vsrai}(left_3, 24)$
13: **return** $vec_{q0}, vec_{q1}, vec_{q2}, vec_{q3}$

Similar to packing algorithm, unpacking algorithm uses special masks, *restore_lo_mask* and *restore_hi_mask*, to shuffle 8-bit integers in a specific order, so, the fully unpacked vectors will be in the correct order. We define these masks as

$$
\begin{aligned}
&\mathit{restore_lo_mask} = \mathbf{vset_epi8}(\\
&\qquad 0, 8, -128, -128, 1, 9, -128, -128, \\
&\qquad 2, 10, -128, -128, 3, 11, -128, -128, \\
&\qquad -128, -128, 4, 12, -128, -128, 5, 13, \\
&\qquad -128, -128, 6, 14, -128, -128, 7, 15 \\
&), \\
&\mathit{restore_hi_mask} = \mathbf{vset_epi8}(\\
&\qquad -128, -128, 0, 8, -128, -128, 1, 9, \\
&\qquad -128, -128, 2, 10, -128, -128, 3, 11, \\
&\qquad 4, 12, -128, -128, 5, 13, -128, -128, \\
&\qquad 6, 14, -128, -128, 7, 15, -128, -128 \\
&).
\end{aligned}
\tag{B.2}
$$

Bibliography

[1] Charu C. Aggarwal. *Neural Networks and Deep Learning A Textbook*. Springer Cham, 1 edition, August 2018.

[2] Brett W. Bader and Tamara G. Kolda. Efficient matlab computations with sparse and factored tensors. *SIAM Journal on Scientific Computing*, 30(1):205–231, 2008.

[3] Yoshua Bengio, Nicholas Léonard, and Aaron Courville. Estimating or propagating gradients through stochastic neurons for conditional computation. *arXiv preprint arXiv:1308.3432*, 2013.

[4] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. *CoRR*, abs/2005.14165, 2020.

[5] Yaohui Cai, Zhewei Yao, Zhen Dong, Amir Gholami, Michael W. Mahoney, and Kurt Keutzer. Zeroq: A novel zero shot quantization framework. *CVPR*, 2020.

[6] Xiaohan Chen, Yu Cheng, Shuohang Wang, Zhe Gan, Zhangyang Wang, and Jingjing Liu. Earlybert: Efficient bert training via early-bird lottery tickets. *arXiv preprint arXiv:2101.00063*, 2020.

[7] Brian Chmiel, Ron Banner, Elad Hoffer, Hilla Ben Yaacov, and Daniel Soudry. Logarithmic unbiased quantization: Simple 4-bit training in deep learning. *arXiv preprint arXiv:2112.10769*, 2021.

[8] Jungwook Choi, Pierce I-Jen Chuang, Zhuo Wang, Swagath Venkataramani, Vijayalakshmi Srinivasan, and Kailash Gopalakrishnan. Bridging the accuracy gap for 2-bit quantized neural networks (QNN). *CoRR*, abs/1807.06964, 2018.

[9] Jungwook Choi, Swagath Venkataramani, Vijayalakshmi (Viji) Srinivasan, Kailash Gopalakrishnan, Zhuo Wang, and Pierce Chuang. Accurate and efficient 2-bit quantized neural networks. In A. Talwalkar, V. Smith, and M. Zaharia, editors, *Proceedings of Machine Learning and Systems*, volume 1, pages 348–359, 2019.

[10] Intel Corporation. Intel® core™ i7-1068ng7 processor. Available: https://ark.intel.com/content/www/us/en/ark/products/196593.html.

[11] NVIDIA Corporation. Introducing geforce rtx 40 series gpus — geforce news — nvidia. Available: https://www.nvidia.com/en-us/data-center/h100/, 2022.

[12] NVIDIA Corporation. Nvidia h100 tensor core gpu. Available: https://www.nvidia.com/en-us/data-center/h100/, 2022.

[13] Neural Magic Developer. Torchvision.models. Available: https://sparsezoo.neuralmagic.com/models/resnet_v1-50-imagenet-pruned90?comparison=resnet_v1-50-imagenet-base, 2023.

[14] Intel Developers. Fp16 half-precision floating-point arithmetic functions. Available: https://www.intel.com/content/www/us/en/docs/programmable/683037/21-2/fp16-half-precision-floating-point-arithmetic.html, 2023.

[15] Neural Magic Developers. Sparsezoo. Available: https://sparsezoo.neuralmagic.com/, 2023.

[16] PyTorch Developers. Pytorch profiler. Available: https://pytorch.org/tutorials/recipes/recipes/profiler_recipe.html.

[17] PyTorch Developers. Torchvision.models. Available: https://pytorch.org/vision/0.8/models.html.

[18] SciPy Developers. Dok matrix. Available: https://docs.scipy.org/doc/scipy/reference/generated/scipy.sparse.dok_matrix.html.

[19] Erich Elsen, Marat Dukhan, Trevor Gale, and Karen Simonyan. Fast sparse convnets. *CVPR*, pages 14629–14638, 2020.

[20] Angela Fan, Edouard Grave, and Armand Joulin. Reducing transformer depth on demand with structured dropout. *International Conference on Learning Representations*, 2020.

[21] Amir Gholami, Sehoon Kim, Zhen Dong, Zhewei Yao, Michael W. Mahoney, and Kurt Keutzer. A survey of quantization methods for efficient neural network inference. *Low-Power Computer Vision*, pages 291–326, 2022.

[22] Gregory Griffin, Alex Holub, and Pietro Perona. Caltech-256 object category dataset. 2007.

[23] Masafumi Hagiwara. A simple and effective method for removal of hidden units and weights. *Neurocomputing*, 6(2):207 – 218, 1994.

[24] Song Han, Jeff Pool, John Tran, and William J. Dally. Learning both weights and connections for efficient neural networks. *Advances in neural information processing systems*, 28, 2015.

[25] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction, Second Edition*. Springer New York, NY, 2 edition, August 2009.

[26] Torsten Hoefler, Dan Alistarh, Tal Ben-Nun, Nikoli Dryden, and Alexandra Peste. Sparsity in deep learning: Pruning and growth for efficient inference and training in neural networks. *The Journal of Machine Learning Research*, 22(1):10882–11005, 2021.

[27] IEEE. Ieee standard for floating-point arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pages 1–84, 2019.

[28] Wenzel Jakob, Jason Rhinelander, and Dean Moldovan. pybind11 – seamless operability between c++11 and python, 2017. https://github.com/pybind/pybind11.

[29] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, December 2014.

[30] Andreas Krause. Lecture notes in introduction to machine learning: Neural networks, March 2021.

[31] Jonathan Krause, Michael Stark, Jia Deng, and Li Fei-Fei. 3d object representations for fine-grained categorization. In *2013 IEEE International Conference on Computer Vision Workshops*, pages 554–561, 2013.

[32] Raghuraman Krishnamoorthi. Quantizing deep convolutional networks for efficient inference: A whitepaper. *arXiv preprint arXiv:1806.08342*, 2018.

[33] Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. Cifar-10 (canadian institute for advanced research).

[34] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, page 1097–1105, 2012.

[35] François Lagunas, Ella Charlaix, Victor Sanh, and Alexander M. Rush. Block pruning for faster transformers. *arXiv preprint arXiv:2109.04838*, 2021.

[36] Yann LeCun, Leon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[37] Daniel Lemire. The xorshift128+ random number generator fails bigcrush. https://lemire.me/blog/2017/09/08/the-xorshift128-random-number-generator-fails-bigcrush/, 2017.

[38] Daniel Lemire, Oleksandr Frei, Travis Downs, and Gareth Jones. Simdxorshift. https://github.com/lemire/SIMDxorshift, 2016.

[39] Jiajia Li, Jimeng Sun, and Richard Vuduc. Hicoo: Hierarchical storage of sparse tensors. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 238–252, 2018.

[40] Jiaoda Li, Ryan Cotterell, and Mrinmaya Sachan. Differentiable subset pruning of transformer heads. *Transactions of the Association of Computational Linguistics*, 2021.

[41] Tailin Liang, John Glossner, Lei Wang, Shaobo Shi, and Xiaotong Zhang. Pruning and quantization for deep neural network acceleration: A survey. *Neurocomputing*, 461:370–403, 2021.

[42] Bangtian Liu, Chengyao Wen, Anand D. Sarwate, and Maryam Mehri Dehnavi. A unified optimization approach for sparse tensor operations on gpus. In *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 47–57, 2017.

[43] George Marsaglia. Xorshift rngs. *Journal of Statistical Software*, 08, 01 2003.

[44] Paul Michel, Omer Levy, and Graham Neubig. Are sixteen heads really better than one? *Advances in Neural Information Processing Systems*, 2019.

[45] Paulius Micikevicius, Dusan Stosic, Patrick Judd, John Kamalu, Stuart Oberman, Mohammad Shoeybi, Michael Siu, Hao Wu, Neil Burgess, Sangwon Ha, Richard Grisenthwaite, Naveen Mellempudi, Marius Cornea, Alexander Heinecke, and Pradeep Dubey. Fp8 formats for deep learning. *arXiv preprint arXiv:2209.05433*, 2022.

[46] Markus Nagel, Marios Fournarakis, Rana Ali Amjad, Yelysei Bondarenko, Mart van Baalen, and Tijmen Blankevoort. A white paper on neural network quantization. *arXiv preprint arXiv:2106.08295*, 2021.

[47] Mahdi Nikdan, Tommaso Pegolotti, Eugenia Iofinova, Eldar Kurtic, and Dan Alistarh. Sparseprop: Efficient sparse backpropagation for faster training of neural networks, 2023.

[48] Badreddine Noune, Phil Jones, Daniel Justus, Dominic Masters, and Carlo Luschi. 8-bit numerical formats for deep neural networks. *arXiv preprint arXiv:2206.02915*, 2022.

[49] OpenAI. Gpt-4 technical report, 2023.

[50] Sinno Jialin Pan and Qiang Yang. A survey on transfer learning. *IEEE Transactions on Knowledge and Data Engineering*, 22(10):1345–1359, October 2010.

[51] Alexandra Peste, Eugenia Iofinova, Adrian Vladu, and Dan Alistarh. Ac/dc: Alternating compressed/decompressed training of deep neural networks. *Advances in neural information processing systems*, 34:8557–8570, 2021.

[52] Sai Prasanna, Anna Rogers, and Anna Rumshisky. When bert plays the lottery, all tickets are winning. *Empirical Methods in Natural Language Processing*, page 3208–3229, 2020.

[53] Sebastian Ruder. An overview of gradient descent optimization algorithms. *CoRR*, abs/1609.04747, 2016.

[54] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 323:533–536, October 1986.

[55] Yousef Saad. Sparskit: A basic tool kit for sparse matrix computation. Available: https://www-users.cse.umn.edu/~saad/software/SPARSKIT/, 1990.

[56] Hassan Sajjad, Fahim Dalvi, Nadir Durrani, and Preslav Nakov. Poor man's bert: Smaller and faster transformer models. *arXiv preprint arXiv:2004.03844*, 2020.

[57] Shaden Smith and George Karypis. Tensor-matrix products with a compressed sparse tensor. In *Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms*, IA3 '15, New York, NY, USA, 2015. Association for Computing Machinery.

[58] Alen Stojanov, Tyler Michael Smith, Dan Alistarh, and Markus Püschel. Fast quantized arithmetic on x86: Trading compute for data movement. In *2018 IEEE International Workshop on Signal Processing Systems (SiPS)*, pages 349–354, 2018.

[59] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel S. Emer. Efficient processing of deep neural networks: A tutorial and survey. *CoRR*, abs/1703.09039, 2017.

[60] Vincent Vanhoucke, Andrew Senior, and Mark Z. Mao. Improving the speed of neural networks on cpus. In *Deep Learning and Unsupervised Feature Learning Workshop, NIPS 2011*, 2011.

[61] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272, 2020.

[62] Elena Voita, David Talbot, Fedor Moiseev, Rico Sennrich, and Ivan Titov. Analyzing multi-head self-attention: Specialized heads do the heavy lifting, the rest can be pruned. *Association for Computational Linguistics*, page 5797–5808, 2019.

[63] Václav Volhejn. Accelerating neural audio synthesis. Master's thesis, ETH Zürich, 2022.

[64] Karl Weiss, Taghi M. Khoshgoftaar, and DingDing Wang. A survey of transfer learning. *Journal of Big Data*, 3(9), May 2016.

[65] Simon Wiedemann, Temesgen Mehari, Kevin Kepp, and Wojciech Samek. Dithered backprop: A sparse and quantized backpropagation algorithm for more efficient deep neural network training. *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops*, pages 720–721, 2020.

[66] Mengzhou Xia, Zexuan Zhong, and Danqi Chen. Structured pruning learns compact and accurate models. *arXiv preprint arXiv:2204.00408*, 2022.

[67] Zhewei Yao, Linjian Ma, Sheng Shen, Kurt Keutzer, and Michael W Mahoney. Mlpruning: A multilevel structured pruning framework for transformer-based models. *arXiv preprint arXiv:2105.14636*, 2021.

[68] Jiecao Yu, Andrew Lukefahr, David Palframan, Ganesh Dasika, Reetuparna Das, and Scott Mahlke. Scalpel: Customizing dnn pruning to the underlying hardware parallelism. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pages 548–560, 2017.

[69] Hantian Zhang, Jerry Li, Kaan Kara, Dan Alistarh, Ji Liu, and Ce Zhang. Zipml: Training linear models with end-to-end low precision, and a little bit of deep learning. In *Proceedings of the 34th International Conference on Machine Learning - Volume 70*, ICML'17, page 4035–4043. JMLR.org, 2017.

[70] Michael Zhu and Suyog Gupta. To prune, or not to prune: exploring the efficacy of pruning for model compression. *arXiv preprint arXiv:1710.01878*, 2017.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Quantized Backpropagation for Sparse Transfer Learning

Authored by (in block letters):
For papers written by groups the names of all authors are required.

Name(s):	First name(s):
Egeli	Berke

With my signature I confirm that
- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date

Zürich, Switzerland
17.09.2023

Signature(s)

B. E.

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.