Towards Efficient Forward Propagation on Resource-Constrained Systems

Günther Schindler[1], Matthias Zöhrer[2], Franz Pernkopf[2], and Holger Fröning[1]

[1] Institute of Computer Engineering
Ruprecht Karls University, Heidelberg, Germany
{guenther.schindler, holger.froening}@ziti.uni-heidelberg.de
[2] Signal Processing and Speech Communication Laboratory
Graz University of Technology, Austria
{matthias.zoehrer, pernkopf}@tugraz.at

Abstract. In this work we present key elements of DeepChip, a framework that bridges recent trends in machine learning with applicable forward propagation on resource-constrained devices. Main objective of this work is to reduce compute and memory requirements by removing redundancy from neural networks. DeepChip features a flexible quantizer to reduce the bit width of activations to 8-bit fixed-point and weights to an asymmetric ternary representation. In combination with novel algorithms and data compression we leverage reduced precision and sparsity for efficient forward propagation on a wide range of processor architectures. We validate our approach on a set of different convolutional neural networks and datasets: ConvNet on SVHN, ResNet-44 on CIFAR10 and AlexNet on ImageNet. Compared to single-precision floating point, memory requirements can be compressed by a factor of 43, 22 and 10 and computations accelerated by a factor of 5.2, 2.8 and 2.0 on a mobile processor without a loss in classification accuracy. DeepChip allows trading accuracy for efficiency, and for instance tolerating about 2% loss in classification accuracy further reduces memory requirements by a factor of 88, 29 and 13, and speeds up computations by a factor of 6.0, 4.3 and 5.0.

1 Introduction

Deep Neural Networks (DNNs) are widely used for many applications including object recognition [16], speech recognition [12] and robotics [17]. While DNNs deliver excellent prediction performance on many classification tasks, they come at the cost of extremely high computational complexity and memory requirements. Both are scarce for resource-constrained systems like mobile processors, which due to their ubiquity otherwise would be prime target for classification tasks. However, DNNs are typically over-parameterized, which motivates various recent efforts in academia and industry to reduce this redundancy and deploy them on embedded systems.

It seems natural to improve resource efficiency by removing this redundancy of over-parameterized neural networks. There are mainly three approaches to remove redundant information within network topologies: the first main approach

is to design novel network topologies (e.g. SqueezeNet [14]), which leverage less parameter to reduce computational complexity and memory requirements. This approach, however, requires expert knowledge in multiple disciplines and is considered extremely time consuming. Second, Hinton et al. [11] propose distilling the knowledge in a neural network which transfers knowledge from an ensemble or from a large highly regularized model into a smaller, distilled model. The third main approach is to use quantization (reduce bit-width of weights and activations) and pruning (reduce the number of connections) on state-of-the-art network topologies (e.g. ResNet [10]).

In this work, we focus on quantization and the implications on forward propagation. Particularly, we present a flexible quantizer that is used in our DeepChip framework (Figure 1). DeepChip enables resource-efficient DNNs and easy prototyping for resource-constrained systems.

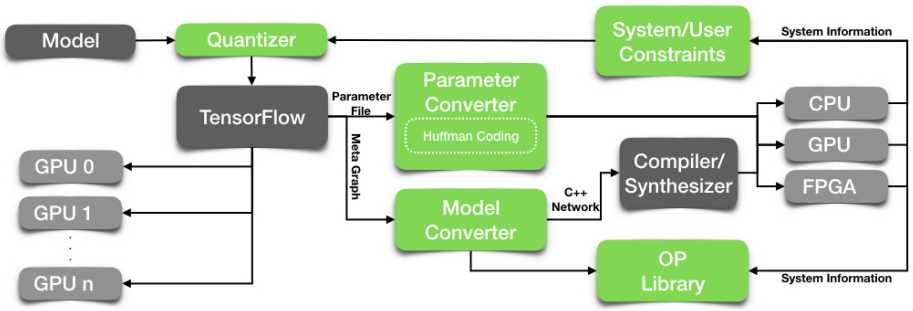

Fig. 1. Overview of the proposed DeepChip framework including design flow and key elements (green: DeepChip, dark grey: third party).

The proposed quantizer uses system and user constraints to optimize a given input model. The parameter and model converters transform the trained model into a compilable source file. The Operator (OP) library includes highly optimized back-ends for a set of different processors. This design implicates several demands on the used quantization strategy:

- **Preserving accuracy:** the quantization needs to achieve full-precision accuracy while still being more efficient in execution. In other terms, only redundancy should be removed, but not non-redundant parameters.
- **Accuracy trade-offs:** real-time requirements and hardware constraints (e.g. on-chip memory capacity) need to be satisfied (possibly at the cost of accuracy loss).
- **Architectural feasibility:** the implementation of forward propagation under the given constraints (real-time, resources) needs to be feasible on a wide range of processor architectures.

In this work, we cover the quantizer, the parameter converter and the core of the OP library. The main contributions of this work are as follows:

- A novel combination of existing quantization strategies that suits modern computer architectures and aims to achieve full-precision accuracy.
- A novel sparse matrix-matrix multiplication algorithm that leverages reduced precision and sparsity in the proposed quantization strategy for efficient forward propagation.
- A novel data structure in combination with compression based on Huffman coding.
- A set of prototypes in the form of convolutional neural networks, which show the potential of the proposed approach on modern computer architectures.

The remainder of this work is structured as follows: related work is reviewed in Section 2, before we reason about our quantization decision in Section 3. Section 4 describes how forward propagation can be implemented on different processor architectures. We report experimental results in Section 5, in combination with a detailed comparison to other approaches. We discuss our approach in Section 6, before we conclude in Section 7. The source code of quantizer and benchmarks is available online [3].

2 Related Work

In this section we briefly review some of the most important methods to quantize weights and activations of a neural network. Binarized Neural Networks (BNNs) were originally introduced by Courbariaux et al. [6] and the basic idea is to constrain both weights and activation to +1 and -1. The BNN approach was then further improved by XNOR-Net [19] and DoReFa-Net [24].

Table 1. Accuracy (Top-1, Top-5) comparison of state-of-the-art quantization approaches of AlexNet on ImageNet for different bit-width combinations of Activations (A) and Weights (W).

A-W		DC	BNN	XNOR	DoReFa	TWN	TTQ	HWGQ
32-32	Top-1	57.2	56.6	56.6	57.2	57.2	57.2	58.5
	Top-5	80.3	80.2	80.2	80.3	80.3	80.3	81.5
32-8/5	Top-1	57.2	–	–	–	–	–	–
	Top-5	80.3	–	–	–	–	–	–
32-2	Top-1	–	–	–	–	54.5	57.5	–
	Top-5	–	–	–	–	76.8	79.7	–
32-1	Top-1	–	–	–	53.9	–	–	–
	Top-5	–	–	–	76.3	–	–	–
2-1	Top-1	–	–	–	–	–	–	52.7
	Top-5	–	–	–	–	–	–	76.3
1-1	Top-1	–	27.9	44.2	45.4	–	–	–
	Top-5	–	50.4	69.2	69.3	–	–	–

[3] `https://github.com/UniHD-CEG/ECML2018`

As can be seen in Table 1, XNOR-Net outperforms BNN by a large margin and DoReFa-Net slightly improves upon XNOR-Net. From an architectural perspective, the inference of these methods can be implemented very efficiently as 1-bit scalar products of two vectors $\mathbf{x}$ and $\mathbf{y}$ of length N can be computed by bit-wise *xnor()* operations, followed by counting the number of set bits:

$$\mathbf{x} \cdot \mathbf{y} = N - 2 \cdot bitcount(xnor(\mathbf{x}, \mathbf{y})), x_i, y_i \in [-1, +1] \forall i \tag{1}$$

Besides lower computational complexity, the memory footprint is reduced as weights require only 1 bit. Using this approach for inference has been proven to be very successful on FPGAs [21], GPUs and ARM CPUs [20]. While the binarization approach only has a negligible accuracy loss to full-precision networks on simple datasets like MNIST or SVHN, binarization on more complex datasets like CIFAR-10 or ImageNet shows severe accuracy degradation.

In order to overcome accuracy degradation, Cai et al. [4] (HWGQ) propose to increase the bit width of activations to 2 bit while still binarizing the weights. This reduces the gap to full-precision networks, for instance there is a 5.8% Top-1 and 5.2% Top-5 accuracy loss for ImageNet. Similar to a 1-bit scalar product, few-bit scalar products can be computed by a bit-serial (or bit-plane) implementation, which uses a bit-wise *and()* operation, followed by population count.

Whereas activations require a relatively high amount of bits to perform as well as full-precision activations, weights seem to require only few bits. For full-precision activations and binary weights, DoReFa has only 3.3% Top-1 and 4.0% Top-5 accuracy loss on ImageNet. For identical activation quantization but ternary weights, Ternary Weight Networks (TWN) [18] achieve 2.7% Top-1 and 3.5% Top-5 accuracy loss. Finally, Trained Ternary Quantization (TTQ) [25] even improves Top-1 accuracy by 0.3% and looses only 0.5% Top-5 accuracy.

Less extreme quantization approaches use 8-bit integer or fixed point representations for activations and weights in order to avoid accuracy loss. The work by Benoit et al. [15] lowers the inference to integer-only computations based on 8-bit matrix-matrix multiplications and is employed as part of TensorFlow [2].

Deep Compression [9] is a compression approach that uses a combination of pruning, trained quantization and Huffman coding. It reduces storage requirements of neural networks significantly without a loss of accuracy. Han et al. [8] showed that this approach can be implemented efficiently on a specialized processor (ASIC). However, the combination of 5-/8-bit weights and sparsity poorly matches general-purpose processors.

3 Quantization

Our quantizer is based on 8-bit activations and 2-bit weights. We quantize weights based on the inspiring concept of TTQ [25] and use fixed-point quantization for the activations. Using this concept, we second the opinion of [4]: while it is possible to quantize weights successfully with only a few bits, activations favor more bits for large-scale classification tasks. Additionally, we show in Section 5 that weight and activation quantization do not interfere each other.

3.1 Quantization of Weights

TTQ uses two scaling coefficients W_l^p and W_l^n for positive and negative weights in each layer l. Both coefficients are independent, asymmetric parameters and are trained together with other parameters (further explained in [25]). The quantized weights w_l^i are calculated by thresholding the full-precision weights on the basis of $\pm\Delta_l$ and are set to either W_l^p, 0 or W_l^n:

$$w_l^i = \begin{cases} W_l^p : w_l > \Delta_l \\ 0 : |w_l| \leq \Delta_l \\ -W_l^n : w_l < -\Delta_l \end{cases} \tag{2}$$

The value of $\pm\Delta_l$ is calculated by using the maximum absolute value of the weights and the hyper-parameter t:

$$\Delta_l = t \cdot max(|w|) \tag{3}$$

We control the sparsity within the weight matrix in our experiments by tuning the threshold hyper-parameter $t \in [0, 1]$.

3.2 Quantization of Activations

Fixed-point representation is superior to integer representation when quantizing activations, because activations are probabilities in the interval of $[0, 1]$ and, thus, would need scaling when implemented with integer representation. Contrary, a fixed-point representation can easily express the probability interval by a variable length fractional part. Fixed-point arithmetic, however, is usually not supported by general-purpose processors and can potentially cause a significant overhead when computed with integer hardware.

We use fixed-point representation to quantize activations and reason in Section 4 why integer hardware is sufficient for our approach. First, we pass the pre-activation $\tilde{a}_i$ through a bounded activation (bounded ReLU) function to ensure that the range of an activation is $a_i \in [0, 1]$:

$$a_i = \begin{cases} 0 : \tilde{a}_i \leq 0 \\ \tilde{a}_i : 0 < \tilde{a}_i < 1 \\ 1 : \tilde{a}_i \leq 1 \end{cases} \tag{4}$$

Next, we adopt the approach of Zhou et al. [24] and use Equation 5 to quantize floating-point activations a_i into k-bit fixed-point activations a_i^q:

$$a_i^q = \frac{1}{2^k - 1} round((2^k - 1)a_i)), \tag{5}$$

where we use 8-bit fixed-point ($k = 8$) for all quantized layers because of two reasons: first, our experiments show that fewer bits cause degradation of classification accuracy on complex datasets. Second, an 8-bit representation is well suited for most computer architectures (further discussions follow in Section 4). However, the bit-width of activations is configurable.

4 Architectural Feasibility

The core operation in the forward propagation of neural networks is the scalar product, which takes two vectors **a** and **b** of the same length and returns a scalar value c. It consists of multiplying connected elements of those two vectors and accumulating the product to the output element:

$$c = \sum_{i=1}^{N} a_i \cdot b_i, \quad a_i, b_i \in \mathbb{R} \quad \forall i \tag{6}$$

In particular, Multiply and Accumulate (MAC) operations are the essence of such a scalar product.

According to Eq. 2, the weight matrix of a layer consists of elements $\{W_l^p, 0, W_l^n\}$. As a consequence, positive and negative weighted values can be treated independently and zero-weighted values can be skipped. As shown in Eq. 7, this results in only two multiplications per scalar product and reduces the major part of the computation to additions.

$$c = W_l^p \cdot \sum_{i \in \mathbf{i}_l^p} a_i + W_l^n \cdot \sum_{i \in \mathbf{i}_l^n} a_i, \quad where \tag{7}$$

$$\mathbf{i}_l^p = \{i | b_i = W_l^p\} \quad \text{and} \quad \mathbf{i}_l^n = \{i | b_i = W_l^n\} \tag{8}$$

This has several advantages since accumulations are extremely efficient (Sec. 4.1) and the underlying scalar-product algorithm can easily exploit sparsity (Sec. 4.2). Furthermore, the constrained range of the weight matrices should allow for a highly effective compression (Sec. 4.3). We leverage these advantages in the following sections for the example of two-dimensional matrices and matrix-matrix multiplication (which build the core of fully-connected layers). Obviously the same approach applies to convolutional layers if they are lowered into matrix multiplications [5].

4.1 Fixed-Point Arithmetic using Integer Hardware

Representing real numbers with fractional part on modern computer architectures is a trade-off between precision and efficiency. The generalized fixed-point representation is $[Q_I.Q_F]$, where Q_I corresponds to the integer part and Q_F corresponds to the fractional part of the number. A common way to leverage integer hardware for inference is based on the fact that activations can be represented by only a fractional part (e.g. $Q0.8$) and weights can be represented by only an integer part (e.g. $Q8.0$). Vanhoucke et al. [22] utilize this approach to accelerate inference on CPUs by using 16-bit integer Fused-Multiply-Accumulate (FMA) instructions:

$$(Q8.8)Accu = (Q8.8)Accu + (Q8.8)((Q0.8)I \cdot (Q8.0)W) \tag{9}$$

$$(int16)Accu = (int16)Accu + (int16)((int8)I \cdot (int8)W) \tag{10}$$

Our approach exploits integer hardware in a similar way, with the difference that we can avoid the multiplication, but instead simply add the 8-bit fractional inputs ($Q0.8$) to a 16-bit ($Q8.8$) accumulator:

$$(Q8.8)Accu = (Q8.8)Accu + (Q0.8)I \tag{11}$$

$$(int16)Accu = (int16)Accu + (int8)I \tag{12}$$

As a result, neither floating-point units nor FMA operations are required, improving the viability of such computations on simple processors like FPGAs and DSPs. Furthermore, integer arithmetic (especially the addition) is faster and more energy efficient than floating-point arithmetic (Table 2).

Table 2. Cycles and energy per MAC and addition (ADD) operation.

Instruction	Cycles (normalized) [3]	Energy (pJ) [13]
float32 FMA	8	4.60
int16 FMA	3	1.60
int16 ADD	1.5	0.05

4.2 Algorithm

The heavy lifting of computing the forward propagation is performed by General-Matrix-Vector-Multiplication (GEMV) or General-Matrix-Matrix-Multiplication (GEMM). Here, GEMM is used for batched inference and GEMV for single-input inference. Whereas sparsity can be exploited rather easily for GEMV, leveraging sparsity for GEMM efficiently is difficult because many code optimizations are not applicable.

In this section, we describe our approach of implementing matrix-vector and matrix-matrix multiplication, which build the core of the Operator (OP) library (see Fig. 1). The basic idea is to split positive and negative scaling coefficients into two separate arrays and to remove zeros (as shown in Eq. 7). We transpose the original weight matrix $\mathbf{W}_l$ into $\mathbf{W}_l^T$ in order to increase data locality during runtime. Now we extract all indices i for which the respective element in the weight matrix $\mathbf{W}_l^T$ is either W_l^p or W_l^n and store the indices in $\mathbf{I}_l^p$ or $\mathbf{I}_l^n$ respectively ($\mathbf{I}_l^p = \{i|w_l^i = W_l^p\}$ and $\mathbf{I}_l^n = \{i|w_l^i = W_l^n\}$). Processing a weight matrix $\mathbf{W}_l$ into $\mathbf{I}_l^p$ and $\mathbf{I}_l^n$ is exemplary illustrated in Eq. 13 and 14.

$$\mathbf{W}_l^T = \begin{pmatrix} 0 & W_l^p & W_l^p & 0 & W_l^n \\ W_l^n & 0 & 0 & W_l^p & 0 \\ W_l^p & W_l^n & 0 & W_l^n & W_l^n \end{pmatrix} \tag{13}$$

$$\mathbf{I}_l^p = \begin{pmatrix} 1\ 2 \\ 3 \\ 0 \end{pmatrix} \quad \text{and} \quad \mathbf{I}_l^n = \begin{pmatrix} 4 \\ 0 \\ 1\ 3\ 5 \end{pmatrix} \tag{14}$$

This transformation is performed after training, by the parameter converter shown in Figure 1, and does not cause additional processing at inference time. Algorithm 1 implements the sparse matrix-matrix multiplication based on Eq. 7.

Algorithm 1 Sparse matrix-matrix multiplication.

Input: $(int8)Input, (int16)\mathrm{I}_l^p, (float32)\mathrm{W}_l^p, (int16)\mathrm{I}_l^n, (float32)\mathrm{W}_l^n$
Output: $(float32)Output$

1: **for** $row := 0$ to $Input.rows()$ **do**
2: **for** $el := 0$ to $\mathrm{I}_l^p[row].elements()$ **do**
3: $index \leftarrow \mathrm{I}_l^p[row][el]$
4: **for** $col := 0$ to $Input.cols()$ **do**
5: $(int16)\mathrm{Accu}_p[col] \leftarrow (int16)\mathrm{Accu}_p[col] + (int8)Input[row][index]$
6: **end for**
7: **end for**
8: **for** $el := 0$ to $\mathrm{I}_l^n[row].elements()$ **do**
9: $index \leftarrow \mathrm{I}_l^n[row][el]$
10: **for** $col := 0$ to $Input.cols()$ **do**
11: $(int16)\mathrm{Accu}_n[col] \leftarrow (int16)\mathrm{Accu}_n[col] + (int8)Input[row][index]$
12: **end for**
13: **end for**
14: **for** $col := 0$ to $Input.cols()$ **do**
15: $(float32)\mathrm{Res}_p \leftarrow castFixedPoint16ToFloat32((int16)\mathrm{Accu}_p[col])$
16: $(float32)\mathrm{Res}_n \leftarrow castFixedPoint16ToFloat32((int16)\mathrm{Accu}_n[col])$
17: $Output[row][col] \leftarrow (float32)\mathrm{W}_l^p \cdot \mathrm{Res}_p + (float32)\mathrm{W}_l^n \cdot \mathrm{Res}_n$
18: **end for**
19: **end for**

As can be seen, the two accumulator arrays $Accu_p$ and $Accu_n$ are used to accumulate the 8-bit fixed-point inputs on the basis of the previously obtained index arrays $\mathbf{I}_l^p$ and $\mathbf{I}_l^n$. Here, $Input$ can be either a vector (batch size = 1) or a matrix (batch size > 1). The accumulations are performed using 16-bit integer addition (as discussed in Section 4.1). We saturate, and therefore approximate, the results of both accumulators to 16 bit since 8 bit might be not sufficient for the integer bits. After the accumulations are computed, the results are cast to single-precision floating point, multiplied with the scaling coefficients W_l^p and W_l^n, and summed up to the final result. The input array is stored in column-major order for batched inputs to increase spatial locality.

We optimized this algorithm for ARM architectures, however, the same optimizations are also applicable to other processors. The algorithm is multithreaded using OpenMP (in Operation 1) without the need of synchronization. Furthermore, the accumulations (in Operation 5 and 11) are vectorized using the ARM NEON processor extension.

4.3 Compression

Memory requirements are usually the limiting factor when deploying neural networks on specialized processors like FPGAs. For instance, models like AlexNet and ResNet-18 require 244MB and 50MB, respectively, while small to mid-sized FPGAs feature only 180kB-4.3MB of on-chip memory (Xilinx Spartan7 series). Being able to stash all parameters on on-chip memory allows using a streaming architecture that can fully utilize computational resources. Furthermore, accessing data from on-chip memory is about 100 times more energy efficient than off-chip memory [13].

In order to reduce memory requirements, we flatten the transposed weight matrices $\mathbf{W}_l^T$, store the signs together with their indices, and apply Huffman coding. Both indices and signs are organized as vectors ($\mathbf{i}_l$ respectively $\mathbf{s}_l$). In $\mathbf{s}_l$, only a single bit per element is required to distinguish between positive and negative scaling coefficient. For the compression, we first determine indices and signs of the scaling coefficients by evaluating all non-zero elements in the weight matrix ($\mathbf{i}_l = \{i | w_l^i \neq 0\}$). Then, we calculate the distance vector $\mathbf{d}_l$ based on the distances between consecutive elements of the previously obtained index vector ($\mathbf{d}_l^j = \mathbf{i}_l^{j-1} - \mathbf{i}_l^j$, with $\mathbf{i}_l^{-1} = 0$). This step decreases the amount of possible values and increases the frequency of appearance of those values. Calculating $\mathbf{s}_l$ and $\mathbf{d}_l$ is exemplary illustrated in Eq. 15, 16, 17 and 18.

$$\mathbf{W}_l^T = \begin{pmatrix} 0 & W_l^p & W_l^p & 0 & W_l^n \\ W_l^n & 0 & 0 & W_l^p & 0 \\ W_l^p & W_l^n & 0 & W_l^n & W_l^n \end{pmatrix} \tag{15}$$

$$\mathbf{s}_l = \begin{pmatrix} 0\ 0\ 1\ 1\ 0\ 0\ 1\ 1\ 1 \end{pmatrix} \tag{16}$$

$$\mathbf{i}_l = \begin{pmatrix} 1\ 2\ 4\ 5\ 8\ 10\ 11\ 13\ 14 \end{pmatrix} \tag{17}$$

$$\mathbf{d}_l = \begin{pmatrix} 1\ 1\ 2\ 1\ 3\ 2\ 1\ 2\ 1 \end{pmatrix} \tag{18}$$

We finally compress $\mathbf{d}_l$ using Huffman coding [1], which leverages the frequency of appearance of values within a given dataset. The principle is to use fewer bits to encode values with a high frequency of appearance. In order to reduce the search space, we use a single codebook that contains the codes for all layers. Then, decompression is simply a matter of looking up the values in the codebook.

5 Experiments

The evaluation of our approach is done on three different convolutional neural networks and datasets: ConvNet [23] on SVHN, ResNet-44 [10] on CIFAR10 and AlexNet [16] on ImageNet. The reported metrics are test accuracy, memory footprint and inference rate in Frames Per Second (FPS).

5.1 Methodology

The inference of convolutional neural networks is usually lowered to matrix-matrix multiplications (fully-connected layer and convolutional layer using the image2col approach). Additionally, for the used models, computations for batch normalization, activation and pooling are insignificant to overall inference rate. Hence, we evaluate throughput by executing all matrix multiplications - required for inference of the respective model - sequentially.

Comparison to related work: we compare these results to a full-precision implementation (baseline), a binarized implementation (BNN) and an 8-bit integer implementation (Int8):

1. The **baseline** implementation uses single-precision floating point for weights and activations. In detail, we rely on the GEMM operator of the Eigen library, as it offers (to the best of our knowledge) the fastest floating-point matrix multiplication on CPUs.
2. DoReFa-Net [24] is selected as representative of **BNNs**. This comparison shows upper bounds regarding inference rate and memory footprint, as it is currently the most effective way (without considering an accuracy drop) to perform inference.We employ an extended version of the Eigen library (from previous work [20]) for such binarized computations.
3. The **Int8** implementation is based on the integer-arithmetic-only approach of Benoit et al. [15] (8-bit activations and weights), omitting reproducing accuracy results as we do not assume a loss in accuracy. We compare against this approach as it is a reliable quantization that is the foundation of mobile inference in TensorFlow Light. Calculations are done using the 8-bit integer GEMM operator of Google's gemmlowp library [4].

In general, the first and last layer of the neural networks are not quantized for DeepChip and BNN, but all hidden layers. Furthermore, all GEMM operations are vectorized and parallelized in order to guarantee a fair comparison. Last, all reported results are based on equal models, including identical hyper-parameters and epochs for training.

5.2 Experimental setup

Training and quantization is performed with Tensorpack [23] on NVIDIA K80 GPUs. As we see an increasing interest in mobile inference using ARM CPUs (TensorFlow Mobile, TensorFlow Light [5] and Baidu's mobile-deep-learning [6]), we use an ARM Cortex-A57 processor to evaluate Algorithm 1 in terms of inference rate. However, note that our approach is not limited to ARM CPUs.

[4] `https://github.com/google/gemmlowp`
[5] `https://www.tensorflow.org/mobile/`
[6] `https://github.com/baidu/mobile-deep-learning`

5.3 ConvNet on SVHN

Training on the SVHN dataset is performed with a single GPU, a batch size of 128 over 200 epochs, and the threshold hyper-parameter set to $t = 0.30$. The achieved results are summarized in Table 3.

Table 3. Results for ConvNet on the SVHN dataset.

	Baseline	BNN	Int8	DeepChip
Accuracy	97.5%	97.0%	–[7]	97.5%
Sparsity	0%	0%	0%	89%
Inference Rate	258 FPS	1,409 FPS	368 FPS	1,337 FPS
Memory Footprint	8,321kB	299kB	2,080kB	192kB

Compared to the baseline implementation, we achieve full-precision accuracy while reducing the memory footprint by a factor of 43 and increase the inference rate by a factor of 5.2. These results are similar to those of BNNs, however with better accuracy and less memory requirements, but a slightly worse inference rate. Also, we significantly outperform the Int8 approach.

5.4 ResNet-44 on CIFAR-10

Training on the CIFAR-10 dataset is performed with two GPUs and a batch size of 128 over 400 epochs. For this dataset, we use parameters of a pre-trained ResNet-44 (full-precision weights) before re-training the neural network again with quantization. The threshold hyper-parameter is set to $t = 0.25$. The results are shown in Table 4.

Table 4. Results for ResNet-44 on CIFAR-10 dataset.

	Baseline	BNN	Int8	DeepChip
Accuracy	92.6%	87.6%	–[7]	92.4%
Sparsity	0%	0%	0%	58%
Inference Rate	69 FPS	532 FPS	100 FPS	191 FPS
Memory Footprint	2,622kB	82kB	655kB	117 kB

Compared to the baseline implementation, at the cost of a slight accuracy loss (-0.2%), DeepChip reduces memory requirements by a factor of 22 and increase the inference rate by a factor of 2.8. By contrast, BNN reduces memory requirements by a factor of 32 and increases the inference rate by 7.7, but results in a significant accuracy loss (-5.0%). Again, DeepChip outperforms Int8 by a large margin. Less sparsity and the accuracy drop of BNN indicate that ResNet on CIFAR-10 is less over-parameterized than ConvNet on SVHN.

5.5 AlexNet on ImageNet

Training on the ImageNet dataset is performed with two GPUs and a batch size of 64 over 100 epochs. The threshold hyper-parameter is set to $t = 0.05$. The achieved results are summarized in Table 5.

Table 5. Results for AlexNet on ImageNet dataset.

	Baseline	BNN	Int8	DeepChip
Top-1 Accuracy	56.2%	45.4%	–[7]	56.4%
Top-5 Accuracy	78.3%	56.4%	–[7]	79.0%
Sparsity	0%	0%	0%	63%
Inference Rate	4 FPS	22 FPS	7 FPS	8 FPS
Memory Footprint	244MB	24MB	61MB	25MB

Compared to the baseline implementation, DeepChip improves memory requirements by a factor of 10, increases the inference rate by a factor of 2.0 and achieves 0.2% better top-1 accuracy. BNN, however, achieves roughly the same improvement regarding memory requirements but increases the inference rate by a factor of 5.5 at the cost of 10.8% top-1 accuracy loss. Even though DeepChip achieves a sparsity of 63% on AlexNet, the improvements over baseline and Int8 approach are less compared to ConvNet or ResNet. This is due to the fact that we currently quantize only hidden layers, but the output layers of AlexNet contributes about 16 MB. Still, we believe that this is not a major issue since input and output layer of more recent networks (e.g. ResNet or Inception) only have a small impact to overall network size.

5.6 Accuracy vs. Efficiency Considerations

In this section we analyze the correlation among quantization strategy, accuracy and efficiency.

Trading Accuracy with Efficiency Using neural networks in resource-constrained systems or under real-time requirements enforces certain demands on memory footprint, latency and inference rate. In this experiment, we increase the threshold hyper-parameter t and evaluate the impact on these metrics.

Increasing the threshold from $t = 0.30$ to $t = 0.35$ on the SVHN dataset results in only 0.7% accuracy loss while the achieved sparsity increases from 89% to 96%. Due to this sparsity increase, the memory footprint is reduced to 95kB and inference rate is increased to 1,538FPS. A similar behavior can be observed for ResNet-44 and AlexNet, as shown in Table 6 respectively Table 7.

[7] We assume, as claimed by the authors [15], no loss in classification accuracy.

Table 6. Results for ResNet-44 on CIFAR-10 for a varying threshold-parameter t.

	$t = 0.25$	$t = 0.30$	$t = 0.35$
Accuracy	92.4%	91.1%	90.1
Sparsity	58%	63%	71%
Inference Rate	191 FPS	255 FPS	293 FPS
Memory Footprint	117 kB	108 kB	90 kB

Table 7. Results for AlexNet on ImageNet for a varying threshold-parameter t.

	$t = 0.05$	$t = 0.10$	$t = 0.15$
Top-1 Accuracy	56.4%	54.7%	53.7%
Top-5 Accuracy	79.0%	77.5%	77.0%
Sparsity	63%	78%	88%
Inference Rate	4 FPS	17 FPS	20 FPS
Memory Footprint	25MB	22MB	19MB

Equalizing Accuracy Quantization using the BNN approach is highly effective in terms of inference rate and memory requirements but can cause accuracy degradation. This degradation can be compensated by either making the neural network wider (increasing the amount of neurons per layer) or deeper (increasing the amount of layers). In this section, we deepen the ResNet model on CIFAR-10 with BNN quantization until the respective quantization roughly reaches full-precision accuracy (ResNet-44 baseline). Our results show, that BNN requires an increase from 44 to 272 layers to achieve 91.2% test accuracy. As a result, DeepChip achieves a memory footprint of 108 kB with 1.5% test accuracy loss (see Table 6), while BNN after scaling results in a footprint of 507 kB with 1.4% test accuracy loss. Furthermore, the additional layers increase training time roughly by a factor of 6.5, compared to baseline ResNet-44.

5.7 Comparing to Deep Compression

In this section, we compare our results with Deep Compression on the example of AlexNet (Table 8). As mentioned earlier, Deep Compression is a highly efficient approach of quantizing and compressing neural networks without any accuracy degradation for specialized processors. As can be seen, Deep Compression outperforms our approach in terms of memory footprint (7.4% better compression ratio) and sparsity (26% more sparsity) on AlexNet. This is mainly because we currently quantize only hidden layers and not input and output layers (the output layer of AlexNet alone has a size of 16 MB). This advantage diminishes when comparing only the quantized layers, e.g. only 1.2% better compression ratio. Still, Deep Compression achieves about 22% more sparsity, which directly translates into fewer operations but relies on 5-/8-bit multiplications. An FPGA synthesis report (Vivado design suite) indicates that for one Processing Element (PE), consisting of 8 channels, and using 8×8 inputs, a multiply accumulate

Table 8. Comparison of Deep Compression and DeepChip (memory footprint and sparsity refer to AlexNet on ImageNet).

	Deep Compression [9]	DeepChip
Top-1 Accuracy	57.2%	56.4%
Top-5 Accuracy	80.3%	79.0%
Memory Footprint (all layers)	6.9 MB	25.2 MB
Memory Footprint (hidden layers only)	5.9 MB	8.7 MB
Compression Ratio (all layers)	97.1%	89.7%
Compression Ratio (hidden layers only)	97.4%	96.2%
Sparsity (all layers)	89%	63%
Sparsity (hidden layers only)	90%	68%
Energy costs per PE (using data from [13])	153.6pJ	15.4pJ
CLB LUTs per PE	765	64
Maximum frequency of PEs	384MHz	400MHz

unit requires roughly 10 times more Configurable Logic Blocks (CLBs) than an accumulator as it is required for DeepChip.

Furthermore, Deep Compression is targeting real-time inference in which batching is usually not used (batch size of 1). Han et al. [9] report excellent speedups for Deep Compression when comparing to dense and sparse GEMV. However, when using batched inputs, sparse GEMV performs roughly equal or even worse than dense GEMM. On the contrary, we target batched inference as well as real-time inference, as efficient processing of frame sequences is of fundamental importance for many applications.

6 Discussion

The concept of ternary weights and 8-bit fixed-point activations, in combination with DeepChip's inference architecture achieve promising performance and is highly flexible. Of particular importance is the fact that redundancy is removed without a loss in test accuracy, and that the concept and architecture are based on a generic processor. As a result, we expect an efficient application to a variety of processor architectures, in particular because our architecture results in a very low amount of rather simple computations (see Section 5.7).

Currently, we do not quantize input and output layers of the neural networks, because quantizing the weights of these layers in the same way as we quantize the hidden layers would result in accuracy drops. Instead, we are considering using either integer or more than just two scaling coefficients for the input and output layer. The sparse matrix-multiplication algorithm and compression approach is also applicable to multiple scaling coefficients. This might be relevant for Recurrent Neural Network (RNNs) too, as related work [7] reports that RNNs require more bits for weight representations than CNNs.

We performed all experiments using 8 bit fixed-point activation representations, because 8 bit suits general-purpose processors well and fewer bits cause

accuracy degradation on complex datasets. The negligible accuracy loss of BNNs on SVHN, however, indicates that efficient quantization is dependent on data set complexity. Even though we did not evaluate fewer bits for activations, we anticipate that this will further improve computations on specialized processors like FPGAs.

7 Conclusion

We have introduced key elements of DeepChip, a framework that targets efficient inference on resource-constrained computing systems and is flexible/configurable for various quantizations, different target architectures, and allows for trade-offs of complexity, accuracy and efficiency. The quantizer generates variable-length fixed-point activations and asymmetric ternary weights. A combination of a simple but highly efficient matrix-multiplication algorithm and compression technique exploits this representation by leveraging reduced precision as well as sparsity to improve memory footprint, latency and inference rate. DeepChip is able to achieve single-precision floating point accuracy while removing redundancy of neural networks. Furthermore, trading accuracy with efficiency allows to satisfy real-time, system and user constraints. Finally, a detailed comparison to other approaches highlights its efficiency and potential regarding processor-agnostic implementations.

Acknowledgments

We gratefully acknowledge the valuable contributions of Andreas Kugel and Andreas Melzer. We also acknowledge funding by the German Research Foundation (DFG) under the project number FR3273/1-1 and the Austrian Science Fund (FWF) under the project number I2706-N31.

References

1. A. Huffman, D.: A method for the construction of minimum-redundancy codes 11, 91–99 (02 2006)
2. Abadi, M.: Tensorflow: Large-scale machine learning on heterogeneous distributed systems. CoRR abs/1603.04467 (2016)
3. ARM: Cortex-a9 neon media - technical reference manual. Tech. rep. (2008)
4. Cai, Z., He, X., Sun, J., Vasconcelos, N.: Deep learning with low precision by half-wave gaussian quantization. CoRR abs/1702.00953 (2017)
5. Chetlur, S., Woolley, C., Vandermersch, P., Cohen, J., Tran, J., Catanzaro, B., Shelhamer, E.: cudnn: Efficient primitives for deep learning. CoRR abs/1410.0759 (2014)
6. Courbariaux, M., Bengio, Y.: Binarynet: Training deep neural networks with weights and activations constrained to +1 or -1. CoRR (2016)
7. Han, S., Kang, J., Mao, H., Hu, Y., Li, X., Li, Y., Xie, D., Luo, H., Yao, S., Wang, Y., Yang, H., Dally, W.J.: ESE: efficient speech recognition engine with compressed LSTM, on FPGA. CoRR abs/1612.00694 (2016)

8. Han, S., Liu, X., Mao, H., Pu, J., Pedram, A., Horowitz, M.A., Dally, W.J.: EIE: efficient inference engine on compressed deep neural network. CoRR abs/1602.01528 (2016)
9. Han, S., Mao, H., Dally, W.J.: Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding. CoRR abs/1510.00149 (2015)
10. He, K., Zhang, X., Ren, S., Sun, J.: Deep residual learning for image recognition. CoRR abs/1512.03385 (2015)
11. Hinton, G., Dean, J., Vinyals, O.: Distilling the knowledge in a neural network. pp. 1–9. NIPS'14 Deep Learning Workshop (03 2014)
12. Hinton, G., Deng, l., Yu, D., E. Dahl, G., Mohamed, A.r., Jaitly, N., Senior, A., Vanhoucke, V., Nguyen, P., Sainath, T., Kingsbury, B.: Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups 29, 82–97 (11 2012)
13. Horowitz, M.: 1.1 computing's energy problem (and what we can do about it) 57, 10–14 (02 2014)
14. Iandola, F.N., Moskewicz, M.W., Ashraf, K., Han, S., Dally, W.J., Keutzer, K.: Squeezenet: Alexnet-level accuracy with 50x fewer parameters and <1mb model size. CoRR abs/1602.07360 (2016)
15. Jacob, B., Skirmantas, K., Chen, B., Menglong, Z., Matthew, T., Andrew, H., Hartwig, A., Kalenichenko, Dmitry: Quantization and training of neural networks for efficient integer-arithmetic-only inference. eprint arXiv:1712.05877 (2017)
16. Krizhevsky, A., Sutskever, I., Hinton, G.E.: Imagenet classification with deep convolutional neural networks. In: Proceedings of the 25th NIPS. pp. 1097–1105. NIPS'12, Curran Associates Inc., USA (2012)
17. Lenz, I.: Deep Learning for Robotics (2016)
18. Li, F., Liu, B.: Ternary weight networks. CoRR abs/1605.04711 (2016)
19. Rastegari, M., Ordonez, V., Redmon, J., Farhadi, A.: Xnor-net: Imagenet classification using binary convolutional neural networks. CoRR abs/1603.05279 (2016)
20. Schindler, G., Mücke, M., Fröning, H.: Linking application description with efficient simd code generation for low-precision signed-integer gemm. UCHPC Workshop, collocated with Euro-PAR (2017)
21. Umuroglu, Y., Fraser, N.J., Gambardella, G., Blott, M., Leong, P.H.W., Jahre, M., Vissers, K.A.: FINN: A framework for fast, scalable binarized neural network inference. CoRR abs/1612.07119 (2016)
22. Vanhoucke, V., Senior, A., Mao, M.Z.: Improving the speed of neural networks on cpus. In: Deep Learning and Unsupervised Feature Learning Workshop, NIPS 2011 (2011)
23. Wu, Y., et al.: Tensorpack (2016)
24. Zhou, S., Ni, Z., Zhou, X., Wen, H., Wu, Y., Zou, Y.: Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients. CoRR abs/1606.06160 (2016)
25. Zhu, C., Han, S., Mao, H., Dally, W.J.: Trained ternary quantization. CoRR (2016)