Term Quantization: Furthering Quantization at Run Time

H. T. Kung*
Harvard University
kung@harvard.edu

Bradley McDanel*
Franklin and Marshall College
bmcdanel@fandm.edu

Sai Qian Zhang*
Harvard University
zhangs@g.harvard.edu

***Index Terms*—Deep neural network (DNN), quantization, accelerator**

***Abstract*—We present a novel technique, called Term Quantization (TQ), for furthering quantization at run time for improved computational efficiency of deep neural networks (DNNs) already quantized with conventional quantization methods. TQ operates on power-of-two terms in expressions of values. In computing a dot-product computation, TQ dynamically selects a fixed number of largest terms to use from values of the two vectors. By exploiting weight and data distributions typically present in DNNs, TQ has a minimal impact on DNN model performance (e.g., accuracy or perplexity). We use TQ to facilitate tightly synchronized processor arrays, such as systolic arrays, for efficient parallel processing. We evaluate TQ on an MLP for MNIST, multiple CNNs for ImageNet and an LSTM for Wikitext-2. We demonstrate significant reductions in inference computation costs (between 3-10×) compared to conventional uniform quantization for the same level of model performance.**

I. Introduction

Deep Neural Networks (DNNs) have achieved state-of-the-art performance across a variety of domains, including Recurrent Neural Networks (RNNs) and Transformers for natural language processing and Convolutional Neural Networks (CNNs) for computer vision. However, the high computation complexity of DNNs makes them expensive to deploy at scale. For instance, in datacenter contexts, a popular model (*e.g.,* email autocomplete [1]) may be queried millions of times per day, with each query requiring 10s to 100s of GFLOPs. Similar concerns are present with edge and endpoint computations.

To address these high computational costs, significant research effort has been spent on developing techniques that reduce the computational complexity of pre-trained DNNs. One of the most commonly used techniques is post-training quantization (see, *e.g.,* [2]), where 32-bit floating-point DNN weights and data (activations) are converted to fixed-point representations (*e.g.,* 8-bit fixed-point) via uniform quantization (UQ). One benefit of post-training quantization is that it does not require access to the original training dataset, and can therefore be applied by a third-party (such as a cloud service or a mobile operator) as a step to reduce costs.

In this paper, we *further quantize* the computation of an already quantized DNN at run time to realize additional computation savings. That is, we propose to perform further run-time quantization on, for example, a 8-bit DNN under UQ while still achieving the same level of model performance. Note that for furthering quantization, we must use new quantization techniques beyond conventional UQ, for otherwise, the original DNN could have been quantized to a lower precision in the first place (*e.g.,* 4-bit UQ instead of 8-bit UQ).

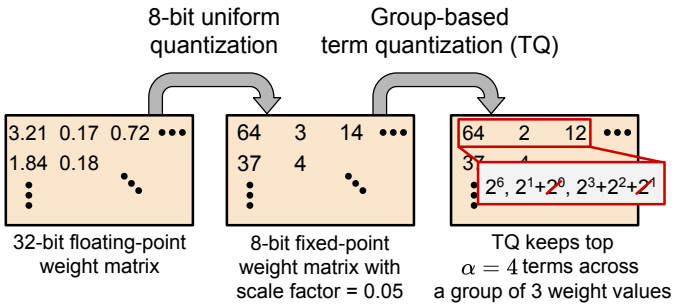

Fig. 1: In conventional quantization, a 32-bit floating point weight matrix (left) is converted to an 8-bit fixed-point format via uniform quantization (middle). We propose to further quantization via Term Quantization (TQ) which is a group-based run-time quantization method (right). By limiting the number of power-of-two terms across a group of values, TQ saves costs and enables a tighter processing bound for DNN dot product computations.

Specifically, we introduce a novel group-based quantization method, which we call *Term Quantization* (TQ). In this work, we define a *term* as a nonzero bit in a quantized fixed-point value. For instance, we say that the 8-bit value 5 (00000101) is composed of two terms: $2^2 + 2^0$. As shown in Figure 1, TQ ranks the terms in a *group* of weight values associated with a dot-product computation to reveal a fixed number of top terms (called a *group budget* or simply term budget) to use for the dot-product computation. By limiting the number of terms to a group budget α and pruning the remaining smaller terms, TQ enables a more efficient implementation of dot-product computations in DNN. **With TQ, the selected terms for a value are based on their relative rankings against terms of other values in the group.** This run-time group-based quantization, a departure from traditional individual value-based quantization such as UQ, allows TQ to carry out additional quantization on an already quantized DNN with a minimum impact to model performance.

*Equal contribution ordered alphabetically.

SC20, November 9-19, 2020, Is Everywhere We Are
978-1-7281-9998-6/20/$31.00 ©2020 IEEE

While allowing further quantization at run time, TQ is able to achieve similar performance as the DNN under conventional uniform quantization (UQ) for two reasons. First, TQ uses group-based term selection, which prunes only smaller terms in a group (*e.g.,* 2^1 and 2^0 terms), leading to minimal added quantization error. A group may have fewer terms than the allocated group budget; in this case no additional truncation needs to be performed. Second, by adapting to the dynamic range of current values in a group, TQ with a small group budget can still discern differences in small weight and data values typically present in DNNs (see Section III-A), where these differences are critical in differentiating features.

With a simple FPGA design, we can use a small number of control bits to reconfigure a hardware supporting quantized computations under both UQ and TQ.

To simplify our introduction of TQ, we use conventional binary representations where all terms are nonnegative. However, shorter *signed-digit representations* (SDRs) which use both positive and negative terms, such as Booth encodings [3], can allow fewer terms in expressing a value and lead to increased computation savings under TQ. To this end, we have developed a new signed encoding called *Hybrid Encoding for Shortened Expressions* (HESE), which in one pass produces SDRs with the theoretical minimum number of terms.

The novel contributions of the paper are:

- The concept of *term quantization* which limits the number of nonzero terms across a group of values up to a term budget α.
- A *group-based term ranking mechanism*, called term revealing which converts values under UQ to TQ.
- A *term MAC* (tMAC) hardware design for the efficient implementation of TQ DNNs.
- An FPGA system which requires *minimal reconfiguration* to efficiently support both UQ and TQ.
- A *one-pass encoding method*, called Hybrid Encoding for Shortened Expressions (HESE), for converting conventional binary representations to minimum-length SDRs. HESE enhances both computation efficiency and performance of TQ.

II. Background and Related Work

In Section II-A, we discuss related work on pruning and quantization techniques for performing efficient DNN inference. Then, in Section II-B, we discuss prior work on hardware architectures which aim to exploit bit-level sparsity. Finally, in Section II-C we illustrate how matrix multiplication is performed with systolic arrays.

A. Pruning and Quantization Methods

There has been significant research efforts in pruning-based methods which exploit value-level sparsity in CNN weights, as performing multiplication with zero operands can be viewed as wasted computation [4]–[12]. However, these pruning methods typically require model retraining, making them infeasible for a third-party that is hosting the model (as it requires access to the training dataset). Additionally, unstructured pruning methods which achieve the best performance (*e.g.,* [4]) are hard to implementing efficiently in special-purpose hardware, as the the remaining nonzero weights are arbitrarily distributed. In this paper, we propose to further reduce the amount of computation even for nonzero values by exploiting bit-level (term) sparsity as opposed to conventional value-level sparsity.

Quantization [13]–[26] lowers precision of the values in weights and data in order to reduce the associated storage, I/O and/or computation costs. However, aggressive post-training quantization under conventional UQ (*e.g.,* from 8-bit to 4-bit) introduces additional error into the computation, leading to decreased model performance. For this reason, many low-precision quantization approaches, such as binary neural networks [27], must be performed during training. Our proposed TQ approach is applied on top of 8-bit uniform quantization (UQ) and does not require additional training. Note that our approach does not reduce the precision (bitwidth) of the weights (*i.e.,* weights are still 8-bit fixed-point values after term quantization) but instead reduces the number of nonzero terms to be used at runtime across a group of weights.

Note the distinction between bitwidth of a value and number of nonzero bits in a value. For example, the value 5 = (0 1 0 0 1) has bitwidth 5 and 2 nonzero bits (terms). TQ aims at reducing the total number of terms used in a group of values rather than the bitwidth for all individual values.

B. Hardware Architectures for Exploiting Bit-level Sparsity

There has been growing interest in exploiting bit-level sparsity (*i.e.,* the zero bits present in the binary representations of weight and data values) as opposed to value-level sparsity discussed in the previous section. A bit-level multiplication with a zero bit can be viewed as wasted computation in the same manner as a value-level multiplication with a zero value, in that both operations do not effect the result. For CNN computations, Bit-Pragmatic introduces an architecture that utilizes a nonzero term-based representation to remove multiplication with zero bits in weights while keeping data in a conventional representation [28]. Bit-Tactical follows up this work by grouping nonzero weight and data values to achieve more efficient scheduling of nonzero computation [29]. Both of these approaches assume 8-bit or 16-bit uniform quantization (*i.e.,* the first step in Figure 1).

However, due to the more fine-grained nature of these bit-level architectures, efficiently scheduling bit-level operations across multiple groups of computations becomes challenging, as each group may have a different amount computation to perform. Generally, there could be stragglers that require significantly more bit-level operations than other groups. Both Bit-Pragmatic and Bit-Tactical handle this straggler problem by adding a synchronization barrier which makes all groups wait until the straggler is finished. Due to this, in processing many groups concurrently, they can only exploit bit-level sparsity up to the degree of the group with most bit-level operations (*i.e.,* the straggler). We find that this worse case can be a factor of 2-3$\times$ more bit-level operations compared to the average case. By comparison, TQ enables a tighter processing bound

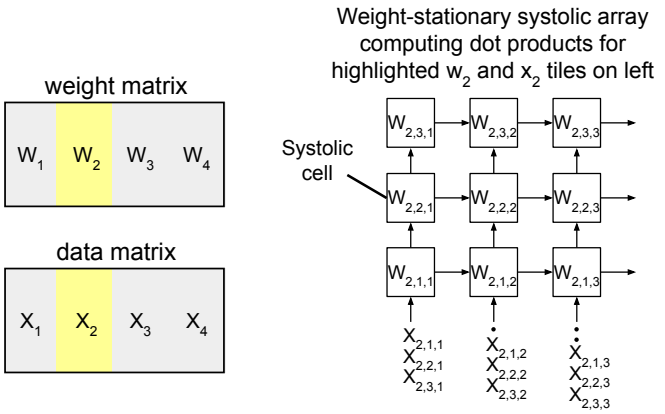

Fig. 2: The weight matrix W and data matrix X for a layer in a DNN (left) are partitioned into four tiles to be processed in a systolic array (right). The highlighted weight tile (W_2) is shown loaded into the systolic array with the data tile (X_2) entering the systolic array from below.

for synchronous computation across all groups. This is done by removing smaller terms from groups with a large number of terms (the second quantization step in Figure 1).

C. Matrix Multiplication and Systolic Arrays

The majority of computation in the forward propagation of a DNN consists of matrix multiplications between a learned weight matrix in each layer and input or data being propagated through the layer, as shown on the left side of Figure 2. Systolic arrays are known to be able to efficiently implement matrix multiplication due to their regular design, dataflow architectures and reduced memory access [30]. The right side of Figure 2 shows a 3×3 matrix multiplication systolic array, for computing dot products between W_2 and X_2 (highlighted in the weight and data matrices). The data in the partition (*e.g.,* $X_{2,1,1}$) are passed into the systolic array from below in a skewed fashion in order to maintain synchronization between cells. We use this systolic array design in our FPGA system in Section V.

III. Term Quantization

In this section, we introduce a group-based quantization method called term quantization (TQ), which is applied to quantized DNNs at run time.

A. DNN Weight and Data Distributions

As mentioned earlier, TQ leverages weight and data distributions of DNNs. DNNs are often trained with weight decay regularization to improve model generalization [31] and batch normalization [32] on data which both improves the stability of convergence and improves the performance of the learned model. A consequence is that the weights are approximately normally distributed and the data follow a half-normal distribution (as ReLU sets negative values to 0). Figure 3 (top row) illustrates these distributions for the weights in 7th convolution layer of ResNet-18 [33] trained on ImageNet [34]

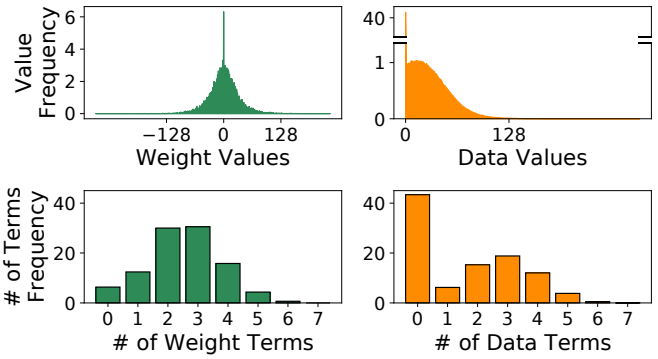

Fig. 3: The distributions of weight and data values (top) shape the distribution of the number of terms in a binary encoding for both weights and data (bottom).

and the data input to the layer. Both the weights and data are quantized to 8-bit fixed-point using uniform quantization (UQ). Other CNN layers exhibit similar distributions for weights. For data, they are the intermediate data from a batch of validation images from ImageNet at layer 7 in ResNet-18.

The higher frequency of small values means that most elements are represented with only 2 or 3 power-of-two terms as shown in Figure 3 (bottom). For instance, the value 6 is represented with two power-of-two terms $(2^2 + 2^1)$. In the figure, 79% of weight values and 84% of data are represented with 3 or fewer power-of-two terms, as opposed to 7 terms in the worst case. Note that the most significant bit (MSB) in the 8-bit representation is used to represent the sign of each value, thus each value has at most 7 terms.

B. Term-pair Multiplications for Dot-product Computation

Assume that we compute dot products in matrix multiplication between quantized weights and data by dividing both vectors into groups of a given length (*e.g.,* 3). Figure 4 illustrates how partial dot products, partitioned into groups of length 3, are computed using term-pair multiplications. In the example, the first value in the data vector $12 = 2^3 + 2^2$ multiplied with the first weight value $2 = 2^1$ is computed using two term-pair multiplications ($2^3 \times 2^1 + 2^2 \times 2^1 = 2^{(3+1)} + 2^{(2+1)} = 16 + 8 = 24$) as shown in Figure 4d. Using this paradigm, we can analyze the number of term-pair multiplications that are required per partial dot product (*e.g.,* with a group size of 3) across all groups in a matrix multiplication.

Figure 5 shows a histogram of the number of term-pair multiplications for partial dot products with groups of 16 values in the 7th convolutional layer of ResNet-18. Interestingly, 99% of these groups require under 110 term-pair multiplications even though the theoretical maximum, where all weight and data values use 7 terms (*i.e.,* every value is $127 = 2^6+2^5+2^4+2^3+2^2+2^1+2^0$), is $16\times7\times7 = 784$. In this work, we propose to restrict the number of term-pair multiplications performed in each partial dot product (*e.g.,* to 110 instead of 784) in order to achieve tightly synchronized parallel processing across systolic

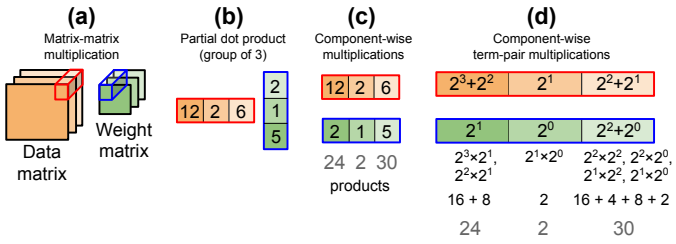

Fig. 4: A matrix multiplication between a weight and data matrix (a) divided into partial dot products of length 3 (b). Each partial dot product is computed via component-wise multiplication (c) which can be performed via component-wise term-pair multiplications (d).

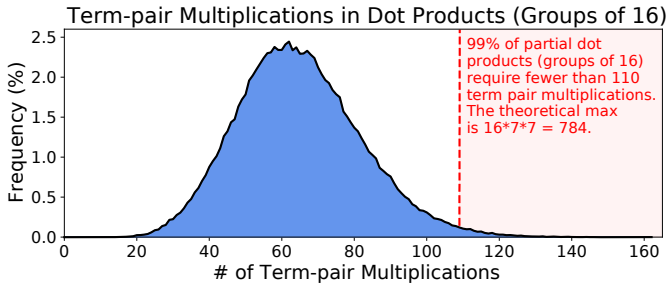

Fig. 5: The number of term-pair multiplications required for partial dot products with groups of size 16 in an 8-bit DNN under the same setup as Figure 3.

cells. We use the number of term-pair multiplications as a proxy for the amount of computation performed during inferences. The hardware system described in Section V performs dot products using this term-pair multiplication approach.

C. Term Revealing for Identifying Terms to Use in TQ

Term revealing is a group-based term ranking method that converts UQ values to TQ values. Term revealing consists of three steps:

1) *Grouping values* as shown in Figure 6 (left). For a given weight matrix, we partition it into equal size groups which are used in dot product computations. The *group size* g, denotes the number of values per group and may assume various values such as 2, 3, 4, 8, 16, etc.
2) *Configure a group budget (i.e., term budget)* α which is the maximum number of terms allowed in a group. The budget bounds the number of terms used in dot-product computations across the values in a group.
3) *Identify top* α *terms* in the group using a *receding water algorithm* that ranks and selects the terms as shown in Figure 6 (right). It keeps the largest α terms in a group and prunes the remaining smaller terms below a waterline. Note that some groups may have fewer than α terms, meaning that no pruning occurs.

Figure 6 illustrate how TQ is applied to a group of $g = 3$ values in a weight matrix with a term budget $\alpha = 4$. The three values are decomposed into their term representations, and

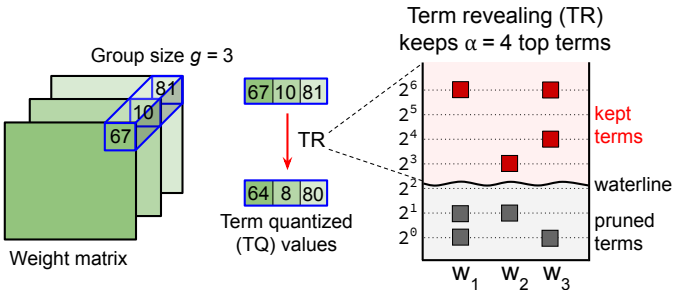

Fig. 6: A weight matrix (left) is partitioned into groups of size 3. The elements of each group (middle) are passed into term revealing (TR). The receding water algorithm (right) based on term ranking keeps the top $\alpha = 4$ terms (red) for the values in the group; the rest of the terms are pruned.

scanned row by row (viewed as a waterline) to reveal terms, starting from the 2^6 term and finishing at the 2^0 term, until the group budget is reached. In the example, the group budget of $\alpha = 4$ is reached at the 2^3 term for w_2. The remaining low-order terms (*e.g.,* 2^2 and below for this group) are pruned, adding a small amount of additional quantization error. For instance, after term revealing, w_3 is quantized from 81 to 80.

Since the position of the waterline is determined by the distribution of terms in a group, the amount of pruning induced by term revealing varies across each group of values. Consider two groups of weights (group a: w_1, w_2, w_3), and (group b: w_4, w_5, w_6) under 7-bit UQ. Figure 7 illustrates the quantization error incurred when 5-bit UQ (truncating the 2^0 and 2^1 terms) and TQ (keeping the top $\alpha = 7$ terms via term revealing) are applied to both groups. For group a, we see that TQ introduces no error as the group has only 7 terms. By comparison, 5-bit UQ introduces error by pruning all of the 2^0 and 2^1 terms, as conventional quantization keeps only the largest 5 terms across all values. For group b, which has significantly more terms, TQ and 5-bit UQ perform a similar amount of truncation. Group b represents a worse case for TQ, as most groups will have significantly fewer terms.

In practice, we can use a small group budget such as $\alpha = 7$ without introducing significant quantization error. By constraining the number of terms to $\alpha = 7$ across the $g = 3$ values, TQ is are able to ensure a tighter processing bound compared to 5-bit UQ. Specifically, assuming each data value has up to 7 term, the maximum number of term pairs with TQ is reduced to $7 \times \alpha = 49$, which is substantially smaller than 5-bit UQ of $7 \times 5 \times 3 = 105$.

D. Reducing Number of Term-pair Multiplications with TQ

We quantify how QT reduces the number term-pair multiplications. Suppose that for a group size of $g = 3$ and the group budget is α the receding water algorithm reveals α_1, α_2 and α_3 number of terms for weight values w_1, w_2 and w_3, respectively, with $\alpha = \alpha_1 + \alpha_2 + \alpha_3$. Suppose further that x_1, x_2 and x_3 have β_1, β_2 and β_3 terms, respectively, with $\beta_i \leq 7$. (For example, $\alpha_1 = 2$, $\alpha_2 = 3$, $\alpha_3 = 1$, and $\beta_1 = 2$,

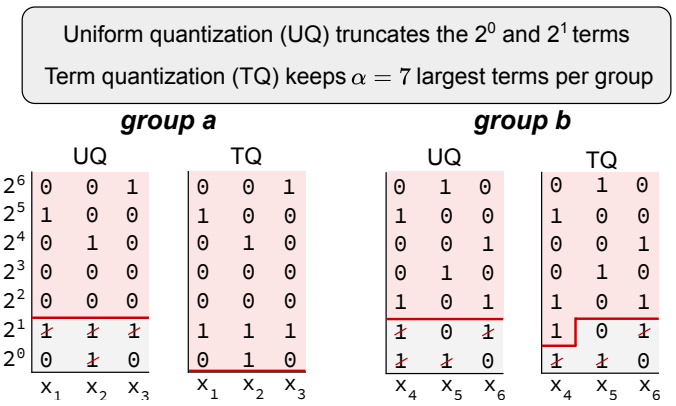

Fig. 7: Applying 5-bit UQ to 7-bit values always truncates smaller terms (*e.g.,* the 2^0 and 2^1 terms). This leads to large quantization error for groups with many small terms as in group a. In contrast, by keeping the top $\alpha = 7$ terms, TQ introduces significantly less quantization error on average. Additionally, TQ reduces the number of term-pair multiplications to $7 \times \alpha = 49$, which is substantially smaller than 5-bit UQ of $7 \times 5 \times 3 = 105$ in the worst case.

$\beta_2 = 4$, $\beta_3 = 3$.) Then, the total number of term pairs to be processed for the dot product computation between w and x is

$$\alpha_1\beta_1 + \alpha_2\beta_2 + \alpha_3\beta_3 \leq 7 \times (\alpha_1 + \alpha_2 + \alpha_3) = 7 \times \alpha$$

In reality, since most weights and data require significantly fewer than the maximum allotted number of power-of-two terms, for most groups, dot products will complete the computation below this bound, as discussed earlier in relation to Figure 5. In this sense, TQ can be viewed as shifting this upper bound from $7 \times 7 \times g$ terms per group in the baseline case (7 terms for both weights and data) to $7 \times \alpha$ terms per group, where $\alpha \ll 7 \times g$ (*e.g.,* $\alpha = 2 \times g$). In Section V, we utilize this significantly reduced upper bound enabled via TQ to implement tightly synchronized processor arrays for DNN inference.

E. Relationship Between Group Size and Group Budget

TQ budgets α terms for a group of size g. Let $\alpha = \mu \times g$ for some μ, where μ is the average number of terms budgeted for each value in the group. Recall from Figure 3 that 79% of weight values are represented in 3 or fewer terms. This means that as the group size g increases, the average number of budgeted terms per value approaches the mean of the weight term distribution. For the weight term distribution in Figure 3, the mean is only 2.46 terms per values, even though some values have as many as 7 terms. Practically, this means that a larger group size allows for a smaller relative term budget α which is close to the mean, as it becomes increasingly unlikely that many groups have more than α terms.

F. Bounding Truncation-induced Error in Dot Products

TQ strives to minimize truncation-induced relative error σ. Suppose that 2^i is the water line determined by TQ under a given group budget. That is, terms smaller than 2^i are truncated. Then, for a group size g and $\mu = 1.5$, kept terms have a total value at least $g \times 2^i + \frac{g}{2} \times 2^{i+1}$, or $g \times 2^{i+1}$, and truncated terms have a total value at most $g \times (2^{i-1} + 2^{i-2} + \cdots + 2^0)$, or $g \times (2^i - 1)$. We have $\sigma = \frac{\text{truncated\_terms}}{\text{kept\_terms}+\text{truncated\_terms}} \leq \frac{\text{truncated\_terms}}{\text{kept\_terms}} \leq \frac{2^i-1}{2^{i+1}} \leq \frac{1}{2}$. Larger μ results in a reduced upper bound on σ.

We provide a bound on the relative error introduced by TQ in truncated dot products between weights and data. For a given group of data (x_1, x_2, x_3), the dot product over the group is $w_1x_1 + w_2x_2 + w_3x_3$ where w_1, w_2 and w_3 are corresponding weights of the filter. Each w_i values may be positive, negative or zero, while x_i data values are non-negative. For simplicity, we assume here that all w_i are positive while noting the result also holds when they are all negative. After TQ, each x_i is replaced with a truncated x_i' in the dot product computation. Let σ_i denote the relative error of x_i' induced by TQ, *i.e.,* $x_i' = x_i(1 - \sigma_i) = x_i - x_i\sigma_i$. Then, the dot product result y with x_i' can be decomposed as follows:

$$\begin{aligned} y &= w_1x_1' + w_2x_2' + w_3x_3' \\ &= w_1(x_1 - x_1\sigma_1) + w_2(x_2 - x_2\sigma_2) + w_3(x_3 - x_3\sigma_3) \\ &= w_1x_1 + w_2x_2 + w_3x_3 - (w_1x_1\sigma_1 + w_2x_2\sigma_2 + w_3x_3\sigma_3) \end{aligned}$$

Therefore, the relative error of the dot product with truncated values as an approximation to the original dot product is:

$$\frac{w_1x_1\sigma_1 + w_2x_2\sigma_2 + w_3x_3\sigma_3}{w_1x_1 + w_2x_2 + w_3x_3}$$

Suppose that, as described above, by TQ we can assure that $\sigma_i \leq \sigma$ for $i = 1, 2, 3$. Then,

$$\frac{w_1x_1\sigma_1 + w_2x_2\sigma_2 + w_3x_3\sigma_3}{w_1x_1 + w_2x_2 + w_3x_3} \leq \frac{w_1x_1\sigma + w_2x_2\sigma + w_3x_3\sigma}{w_1x_1 + w_2x_2 + w_3x_3}$$

Thus, the relative error in the computed dot products $w_1x_1' + w_2x_2' + w_3x_3'$ is bounded by σ.

IV. Hybrid Encoding for Shortened Expressions

In this section, we present Hybrid Encoding for Shortened Expressions (HESE), a signed power-of-two encoding which produces in one pass a signed-digit representation (SDR) with the minimum number of terms for binary input. HESE complements TQ by reducing the number of terms per value in a group before TQ is applied across the group.

A. Signed-digit Representations

Signed-digit representations (SDRs) are a type of positional encoding system, where each position can have a coefficient of $\{-1, 0, 1\}$ as opposed to only $\{0, 1\}$ in a conventional binary encoding. Avizienis proposed the use of SDRs in bit-parallel circuits to remove carry-propagation chains in additions and multiplications [35]. Drake et al. proposed a similar approach involving carry-free addition and subtraction in an optical computing regime [36]. More commonly, Booth radix-4 encoding [3] has been used to convert binary representations with only positive power-of-two terms (*e.g.,* $30 = 2^4 + 2^3 + 2^2 + 2^1$) into shorter representations with both positive and negative power-of-two terms (*e.g.,* $30 = 2^5 - 2^1$).

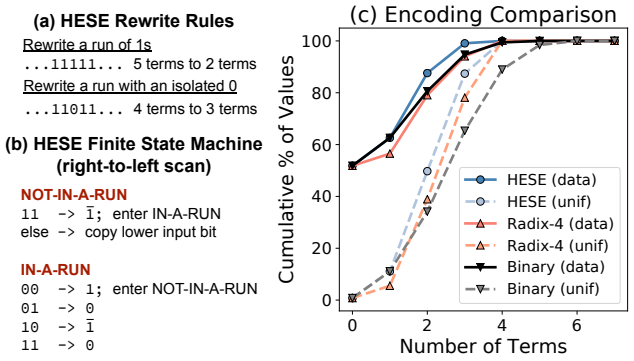

Fig. 8: (a) HESE rewrite rules for converting binary encodings into SDRs. (b) The HESE finite state machine, which operates on two input bits and outputs one digit per state transition. (c) The SDRs generated by HESE require fewer terms than both binary and Booth radix-4 encodings for 8-bit values over DNN data (data) and a uniform distribution (unif).

Booth radix-4 encoding bounds the number of power-of-two terms in an n-bit value to $\frac{n}{2}+1$ [3]. This encoding is utilized in the design of efficient Booth multipliers to provide a smaller bound on the amount of computation required for the multiplication of any pair of n-bit values. This bound is necessary for synchronization purposes across multiple processing elements (*e.g.,* systolic arrays). However, Booth encoding does not lead to the minimum-length SDR in the case of a number with an isolated zero. For instance, 27 (11011 in binary) will be converted into $10\bar{1}10\bar{1}$ in Booth, while the minimum-length encoding is $100\bar{1}0\bar{1}$.[1]

There has been other prior work on developing algorithms to convert binary to minimum-length SDR. Jedwab et al. proposed such an algorithm, with a subsequent proof that the generated SDR has the minimum-possible number of terms [37]. However, the proposed algorithm is not amenable to efficient hardware implementations as written, since it requires multiple passes through the input. In the next section, we present an efficient one-pass two-bit encoding method called HESE for producing SDRs that still achieves the minimum number of terms.

B. Overview of HESE

HESE is a hybrid encoding method that combines Booth, which efficiently handles strings of 1s, with an additional rules for an isolated 0 surrounded by at least two 1s on each side. The first rewrite rule in Figure 8a shows that this encoding reduces a sequence of 5 1s, such as (11111) to only two terms ($10000\bar{1}$). The second rule shows how a sequence with an isolated 0, such as 11011 is rewritten as $100\bar{1}0\bar{1}$, which has only 3 terms. Isolated 1s in the input remain 1s in the output. Figure 8b provides a finite state machine for HESE. It begins in the NOT-IN-A-RUN mode and each state corresponds to two bits in the input sequence. On state transitions, it consumes a single bit of the input and outputs a single signed digit. Transitions from the NOT-IN-A-RUN mode to the IN-A-RUN mode are triggered by observing a 11 in the input (denoting a run of at least two 1s). Likewise, transitions from IN-A-RUN mode back to NOT-IN-A-RUN mode occur when a 00 sequence is observed (or the input is consumed).

The one-pass HESE described in Figure 8b uses right-to-left scan. We can design a similar scheme using left-to-right scan. HESE can also be extended to convert non minimum-length SDRs into minimum-length SDRs. Essentially, by replacing adjacent mixed-sign nonzero terms, $+-$ or $-+$, with a nonzero term and a zero term, we end up with strings of 1s or strings of -1s. Like before, the two rules in Figure 8a can rewrite these strings as well as strings with isolated 0s to derive the minimum-length SDR. However, in this work, we only use HESE to convert binary to minimum-length SDRs.

C. Reducing Number of Terms per Encoding

HESE encodings have strictly equal or fewer terms than binary and Booth radix-4. Figure 8c shows the number of terms required for these encodings across two distributions of values: data values from ResNet-18 and values drawn from a uniform distribution over the same range as the data. The x-axis is the number of terms required to represent a value and the y-axis is the cumulative percentage of values that are represented within a given number of terms. HESE outperforms both Booth and binary across both distributions of values.

As expected, Booth leads to more compact representations than binary for values drawn from the uniform distribution. However, most of the reduction in terms comes from larger values in the 8-bit range (with many 1s), which occur much less frequently for the data, as depicted in Figure 3 (bottom). Therefore, radix-4 performs equal or worse than binary for the distribution of data values we are interested in. By comparison, when applying HESE on data, 99% of values are represented in 3 or fewer terms. Practically, this means in furthering quantization at run time with TQ, we can use only 3 power-of-two terms for both weights and data in the 8-bit range with minimum impact on model performance.

V. Hardware Design for Efficient Term Quantization

In this section, we present a TQ-based hardware design for accelerating DNN computation. Figure 9 provides an overview of our TQ system design. It consists of the following components: (1) weight and data buffers which store DNN layer weights and input/intermediate data, (2) a systolic array which performs dot products between weights and data using term MACs (tMAC) described in Section V-B, (3) a binary stream converter to convert systolic array output into binary representation (Section V-C), (4) a ReLU block (Section V-C), (5) a HESE encoder (Section V-D) to convert the binary representations to shorter SDRs, and (6) a term comparator (Section V-E) which selects the top α terms in a group. In Section V-A, we first give some high-level reasoning on how tMAC can save computation.

[1]Here, $\bar{1}$ represents a term with a negative coefficient, such as -2^2.

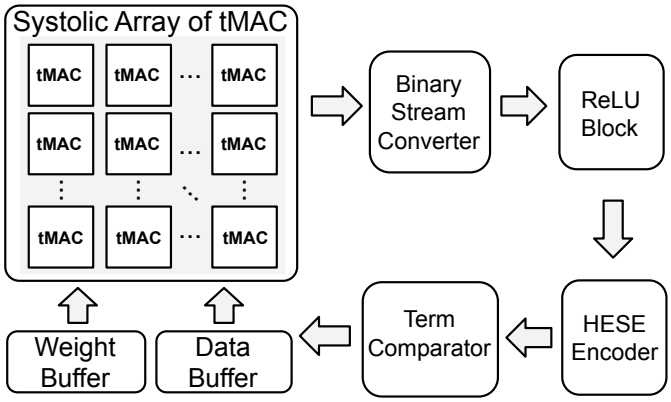

Fig. 9: The term quantization (TQ) system design.

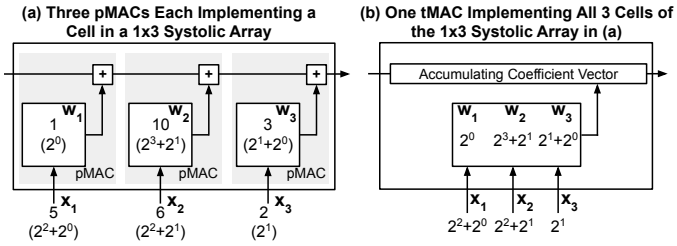

Fig. 10: (a) A 1×3 systolic array. Each of the three systolic cells is based on a conventional bit-parallel MAC (pMAC) which performs an 8-bit multiplication between weights (w) and data (x) values and a 32-bit accumulation each systolic array cycle. (b) The proposed term MAC (tMAC) processes all term-pair multiplications (*e.g.,* $2^2 \cdot 2^1$), for the same systolic array cycle, across a group of weight and data values (group size $g = 3$ here) in a bit-serial fashion. When using a group budget α for weights and at most β terms for each data value, the number of term-pair multiplications is bounded by $\alpha \cdot \beta$. Here, we have 8 term-pair multiplications, and with $\alpha = 6$ and $\beta = 2$, $\alpha \times \beta = 12$. We note that $8 < 12$.

A. High-level Comparison Between Bit-parallel MAC (pMAC) and Term MAC (tMAC)

To help understand the inherent advantage of TQ, we explain how our proposed term MAC (tMAC) saves a significant amount of work over a conventional parallel MAC (pMAC). Here, we define *work* as the amount of computation, including both arithmetic and bookkeeping operations, which are performed per group. The work incurred by a method largely determines the energy, area, and latency of its implementation.

To be concrete, we study a 1×3 1D systolic array of 3 cells, as depicted in Figure 10a, for the processing of groups of 3 data values (x_1, x_2, x_3) in computing their dot products with weights (w_1, w_2, w_3) pre-stored in the systolic array. To provide a baseline for comparison, we consider a conventional implementation, where each cell has a bit-parallel MAC (pMAC) performing an 8-bit multiplication, $w \times x$, and a 32-bit accumulation adding an intermediate y to the computed $w \times x$. In comparison, Figure 10b depicts the proposed term MAC (tMAC) based implementation. We note that tMAC significantly reduces the work by only processing available term pairs. The number of term pairs is relatively small due to high bit-level sparsity generally presented in CNN weights and data. This comparison result applies to a general 2D systolic array, which is a stack of 1D systolic arrays. **In Section VII-A, we show how this analysis translates to realized performance on an FPGA with a group size of $g = 8$.**

For this illustrative analysis, we assume that tMAC uses a TQ group of size $g = 3$ and budget $\alpha = 6$ for weight values, and $\beta = 2$ leading terms for data values under HESE (Section IV-B). Thus, for weights, each value in a group uses on average $\mu = 2$ terms. As we show in Table III, TQ will incur a minimum decrease in classification accuracy (*e.g.,* less than 0.15%) when dropping lower-order terms exceeding the group budget α across multiple CNNs.

Our analysis on work proceeds as follows. A conventional pMAC implementation of a single systolic cell incurs 7 8-bit additions for the multiplication $w \times x$ and 1 32-bit accumulation operation for $y + w \times x$. Therefore, the pMAC implementation of a 1×3 1D systolic array with three cells requires a total of 21 8-bit additions and 3 32-bit accumulation operations. In contrast, a tMAC implementation incurs significantly less work. Specifically, it uses at most 12 3-bit additions on exponents of power-of-two terms (weight and data exponents are both less than 8) for term-pair multiplications. (Recall that we assume $\alpha = 6$ for weight groups and $\beta = 2$ for data values, as depicted in Figure 10b). The updating of accumulating coefficient vector (discussed in detail in the next section) requires bookkeeping operations for bit alignment, etc., with work we assume is no larger than 12 3-bit additions. Thus, tMAC substantially reduces work compared to pMAC, that is, 24 3-bit additions vs. 21 8-bit additions plus 3 32-bit accumulations.

B. Term MAC (tMAC) Design

The term MAC (tMAC) performs dot products between a data and weight vector of group size g by multiplying all term pairs. Figure 11 illustrates how these term-pair multiplications in tMAC are performed for a group of size $g = 4$ and a group budget $\alpha = 8$. TQ ensures that there are 8 or fewer terms across all weight values in the group. For illustration simplicity, assume all data values can be represented with a single term (our implementation allows as many as 3 terms per data value). Figure 11 depicts that 8 term-pair multiplications with results added to a coefficient vector.

The coefficient vector stores the current partial result of the dot product as coefficients for each power-of-two term. In this example, the coefficients are set to (1, 3, -1, 0, 4, 1), which represents a value of $1 \times 2^5 + 3 \times 2^4 - 1 \times 2^3 + 0 \times 2^2 + 4 \times 2^1 + 1 \times 2^0 = 81$. For the first term pair in the figure, $(-2^0, +2^2)$ in $w_1 x_1$, the coefficient for 2^2 is decremented by 1 as the signs of the terms differ. Once all exponent additions are completed for a dot product, the coefficient vector is reduced to a single value. As the largest term is 2^7, assuming 8-bit UQ, the largest

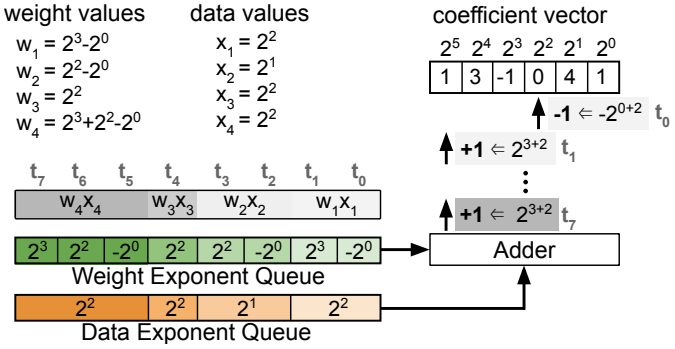

Fig. 11: Term-pair multiplication for a dot product across a group of 4 weight and data values over 8 cycles (t_0 to t_7).

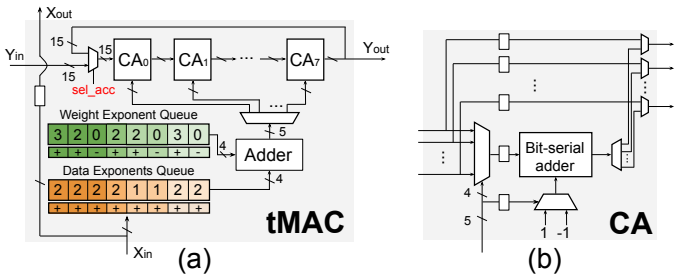

Fig. 12: (a) The term MAC (tMAC) performs term-pair multiplications between data and weight terms for a group of values. (b) A coefficient accumulator (CA) takes the adder result and add or subtract 1 from the corresponding coefficient.

term pair is $2^7 \times 2^7 = 2^{14}$. Therefore, the coefficient vector has a length of 15 in order to store all possible term-pair results from 2^0 to 2^{14}. To prevent overflow for dot products, each element in the coefficient vector has a relatively high bitwidth (12 bits).

The hardware design of tMAC is shown in Figure 12a. The weight and data exponents are stored in queues, with the sign of each term stored in the associated sign queues (one bit per term). For instance, the term -2^2 would store a 2 in the exponent queue and a minus ($-$) in the sign queue. The shaded green and orange colors denote the term-pair boundaries for each data $\times$ weight multiplication. Each cycle, a pair of exponents from these queues are passed into the adder, which computes the sum of the exponents, sets the sign, and sends to the result to a *coefficient accumulator* (CA) (Figure 12b). Therefore, to process a group with 8 term pairs takes 8 cycles in total.

The CAs perform bit-serial addition between the coefficient vector and the output of the exponent adder. Due to the bit-serial design, the number of CAs must match the size of the data and weight queues (8 in this example) in order to maintain synchronization across the cells of the systolic array. At each cycle, one of the eight CAs takes the sum of two exponents from the adder, and adds/subtracts 1 to/from the corresponding coefficient. In our implementation, each tMAC can choose to reuse the current coefficient vector or take the new coefficient vector from its neighboring cell via the selection signal sec_acc, as depicted in Figure 12a.

C. Binary Stream Converter and ReLU Block

The binary stream converter takes a coefficient vector generated by the systolic array and converts it into a single value in a binary format. This is achieved by multiplying each element in the coefficient vector by the corresponding power-of-two term and summing over the partial results. For instance, given a coefficient vector of $(3, -2, 1)$, the binary stream converter first multiplies each coefficient with the corresponding power-of-two term (i.e., $3 \times 2^2 = 12$, $-2 \times 2^1 = -4$, $1 \times 2^0 = 1$). Then, it generates the binary value by summing these partial results (i.e., $12 - 4 + 1 = 9$). The outputs of the binary stream converter are sent to the ReLU block in a bit-serial fashion. Using a two's complement representation for the outputs, the sign can be determined by detecting the most significant bit of the output streams. The ReLU block buffers all the lower bits until the MSB arrives. Then, it outputs zero if the sign of the MSB indicates that the value is negative; otherwise it outputs the original bit stream.

D. HESE Encoder

The HESE encoder produces two bit streams, which represent the magnitude and sign of each power-of-two term, respectively. For example, for a bit-serial input of $31 = 00011111$, the HESE encoder will produce two output streams: 00100001 (magnitude) and 00000001 (sign), to indicate $31 = 2^5 - 2^0$. The HESE encoder is implemented with the finite state machine in Figure 8b.

E. Term Comparator

The term comparator in Figure 13a selects the top α terms from the outputs of every g consecutive HESE encoders, where α and g are the group budget and group size, respectively. Figure 13b shows the operation of term comparator on the outputs of four HESE encoders, the HESE encoder outputs are divided into two groups, where each group has a group size $g = 2$ and group budget of $\alpha = 3$. The inputs enter the term comparator in a reverse order such that their most significant bits (MSB) enter the term comparator first. Each cycle, the term comparator counts the total number of nonzero bits encountered so far, and truncates the remaining low-order terms once the group budget α is reached for a group.

The term comparator contains multiple accumulate and compare (A&C) blocks which are arranged into a tree structure. Each A&C block takes a single input bit stream and counts the total number of nonzero bits in this stream. Figure 14 shows how the A&C blocks can be reconfigured for different group sizes g. For $g = 1$, the A&C blocks on the first level of the tree will compare the number of nonzero bits in their input stream against the group budget α, and truncate each stream accordingly. If the group size is larger than 1 (*e.g.,* 2), each A&C block in the first level of the tree will forward its input stream together with the nonzero bit count to its parent A&C block. The parent A&C block then operates on these two

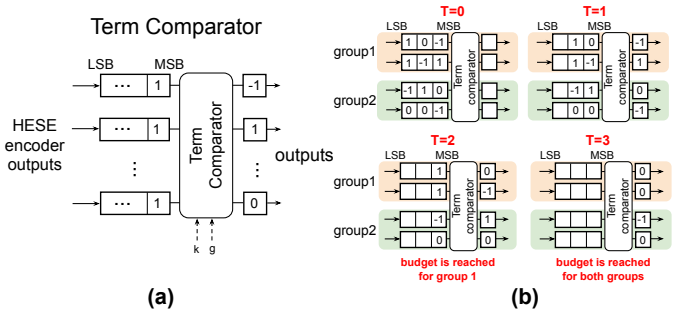

Fig. 13: (a) The design of the term comparator which implements TQ via term revealing. The term comparator takes the group size g and group budget α as inputs, counts the total number of terms within each group, and sets the corresponding terms to zeros once the group budget is reached. (b) An example of term comparator operating on two groups. At T=2, the group budget is reached for group 1 and all the remaining terms are pruned. At T=3, group budget is also reached for group 2.

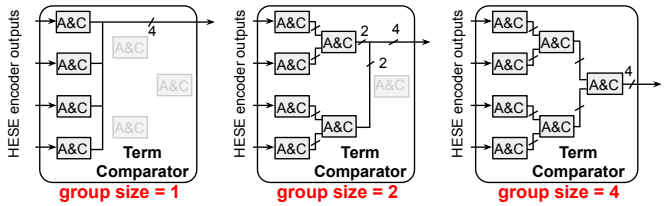

Fig. 14: Configurations of term comparator under different group sizes.

streams in a similar fashion to its children. The tree architecture allows for minimal changes to the term comparator under different group sizes. This design leads to a low reconfiguration overhead and maximum level of hardware reuse.

F. Memory Subsystem

Our memory subsystem consists of a data and weight buffer. The data buffer holds the term exponents and signs for both the input and result data of the current layer, and the weight buffer holds the term exponents and signs of the weights for each group. For the weight buffer, we use double buffering to prefetch the next weight tile from the off-chip DRAM so that the computation of the systolic array can overlap with the data transfer from the off-chip DRAM to weight buffer.

G. FPGA Reconfiguration for UQ and TR

As described in Section VII, the TQ system can be reconfigured for different group sizes g and group budgets α, in order to adapt to dynamic requirements on group size and group budget during inference with a negligible delay. In addition, our system can also support conventional UQ by performing power-of-two operations with binary representations. Since UQ does not require TQ or HESE encoding, the term comparator and HESE encoder can be turned off by using clock gating

TABLE I: Control registers for supporting UQ and TQ.

	Uniform quantization (UQ)	Term quantization (TQ)
HESE_ENCODER_ON (1 bit)	HESE encoder is turned off by setting this bit to 0	HESE encoder is turned on by setting this bit to 1
COMPARATOR_ON (1 bit)	term comparator is turned off by setting this bit to 0	term comparator is turned on by setting this bit to 1
QUANT_BITWIDTH (4 bit)	quantization bitwidth used for UQ	quantization bitwidth used for TQ
DATA_TERMS (4 bit)	same as the quantization bitwidth for UQ	Maximum number of power-of-two terms in data for TQ
GROUP_SIZE (3 bit)	group size is set to 1 for UQ	group size is between 2 to 8 for TQ
GROUP_BUDGET (5 bit)	group budget is the same as quantization bitwidth for UQ	group budget can be up to $8 \times 3 = 24$ for TQ

to reduce power consumption. Table I summarizes all of the control registers which need to be modified when switching between UQ and TQ. The switching process only takes several clock cycles (*i.e.,* within 100ns for our FPGA implementation).

VI. Term Quantization Evaluation

In this section, we evaluate the performance of term quantization (TQ) when applied to an MLP on MNIST [38], a broad range of CNNs (VGG-19 [39], ResNet-18 [33], MobileNet-V2 [40], and EfficientNet-b0 [41]) on ImageNet [34], and an LSTM [42] on Wikitext-2 [43]. In Section VI-A, we compare TQ against conventional uniform quantization (UQ) on the performance (*i.e.,* accuracy or perplexity) of these DNNs. Then, in Section VI-B, we provide analysis on how the α (term budget) and g (group size) parameters impact the classification accuracy. Next, in Section VI-C, we analyze the individual contribution of HESE and TQ on model performance. Finally, in Section VI-D, we show that the quantization error introduced by TQ is substantially less than when using a more aggressive UQ setting (*e.g.,* 6-bit uniform quantization).

To perform this analysis, we have implemented a CUDA kernel for TQ which only increases the inference runtime of a pre-trained model running on a NVIDIA 1080 Ti by under 5%. This means that the validation accuracy for a pre-trained CNN for ImageNet can still be obtained within several minutes. Using pre-trained models has the advantage of making parameter search (*e.g.,* for group size g and term budget k) simple compared to methods such as weight pruning [4] that require model retraining which takes hours or days for each setting. Before applying TQ, each model is quantized from 32-bit floating-point to 8-bit fixed-point using a layerwise procedure described in [44].

A. Comparing Term Quantization to Uniform Quantization

Motivated by the design in Section V, we are interested in minimizing the number of term-pair multiplications per sample, as this directly translates to the processing latency of a sample. For the uniform quantization (UQ) approach with 8-bit fixed-point weights and data, each multiplication translates to $7 \times 7 = 49$ term-pair multiplications. By comparison, for TQ, the number of term-pair multiplications is instead bounded by the average number of terms (μ) which is shared across a group of values. We show that TQ, under varying group sizes g and group budgets alpha, can achieve a significant reduction

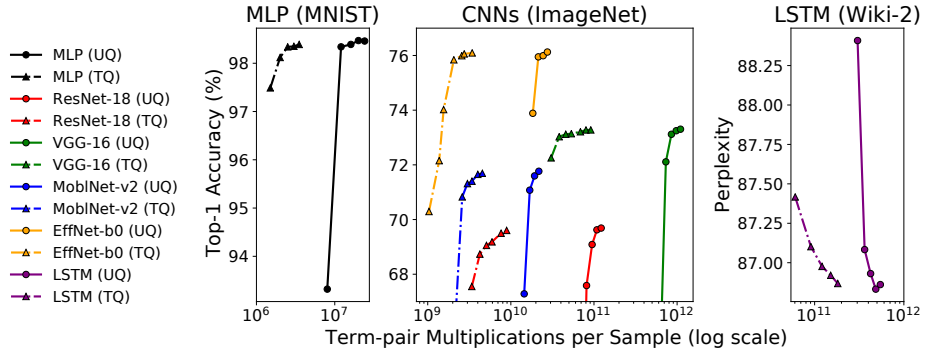

Fig. 15: Comparing uniform quantization (UQ) and term quantization (TQ) for an MLP on MNIST (left), CNNs on ImageNet (center), and an LSTM on Wikitext-2 (right). The UQ settings vary the weight bitwidth (from 4 to 8 bits), while the TQ settings vary g (group size) and α (number of terms per group). TQ reduces the number of term-pair multiplications per sample over UQ by 3-10× across the three types of DNNs.

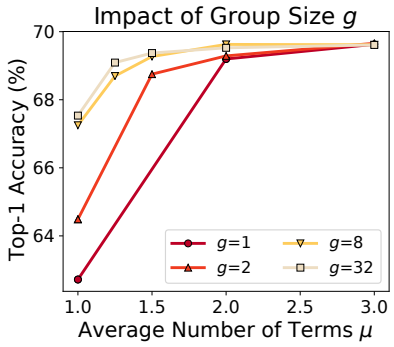

Fig. 16: A larger group size g improves ResNet-18 ImageNet classification accuracy for a given average number of terms μ.

(*e.g.,* 3-10×) over UQ while maintaining the nearly identical performance (*e.g.,* within 0.1% accuracy).

1) MLP on MNIST: We train an MLP with one hidden layer with 512 neurons for MNIST using the parameter settings given in the PyTorch examples for MNIST[2]. Figure 15 (left) shows the performance of UQ and TQ applied to the pre-trained MLP. TQ achieves a 5× reduction in number of term-pair multiplications over UQ while achieving a classification accuracy of 98.4% (compared to the 98.5% baseline).

2) CNNs on ImageNet: We use pre-trained models provided by the PyTorch torchvision package[3] for VGG-16, ResNet-18, and MobileNet-v2 and a PyTorch implementation[4] of EfficientNet. Figure 15 (center) shows the performance of TQ and UQ for the 4 CNNs. TQ achieves a 14× reduction in term-pair multiplications over UQ for VGG-16, which is known to be significantly overprovisioned (*e.g.,* amenable to quantization and pruning). Even for more recent models, which have significantly fewer parameters, such as MobileNet-v2 and EfficientNet-b0, TQ is still able to achieve a 4× and 6× reduction in term-pair multiplications, respectively, losing less than 0.1% classification accuracy compared to the 8-bit UQ settings. Generally, we see that more aggressive TQ settings (*e.g.,* with a reduced group budget) appears to degrade the accuracy more gracefully than more aggressive UQ settings (*e.g.,* with reduced bitwidth for weight).

3) LSTM on WikiText-2: We train a 2-layer LSTM with 650 hidden units (*i.e.,* neurons), a word embedding of length 650, and a dropout rate of 0.5, following the PyTorch word language model example[5]. This baseline model achieves a perplexity of 86.85. Figure 15 (right) shows how the perplexity of the pre-trained model is impacted by UQ and TQ. Again, we find that TQ is able to reduce the number of term-pair multiplications by a significant factor of 3×, while achieving the same perplexity.

[2]https://github.com/pytorch/examples/tree/master/mnist
[3]https://github.com/pytorch/vision/tree/master/torchvision/models
[4]https://github.com/lukemelas/EfficientNet-PyTorch
[5]https://github.com/pytorch/examples/blob/master/word_language_model

B. Improved Term Allocation with Larger Group Size

Figure 16 shows the classification accuracy for ResNet-18 as the average number of terms μ is varied for different group sizes. As the group size increases, the variance in number of terms across values in a group shrinks, meaning that a larger group budget α at a fixed μ ratio is strictly better than a smaller α for the same μ. As observed, the classification accuracy for a larger group size strictly outperforms all settings with smaller group sizes. For instance, a group size of 8 with μ of 1 achieves a classification accuracy of 67.72% which is 5.21% better than a group size of 1 at the same μ value. Note that a group size of 1 is equivalent to truncating each value to exactly μ terms.

C. Isolating the Impact of TQ and HESE

Figure 17 shows the relative impact of TQ and HESE in terms of classification accuracy by measuring them in isolation. The UQ + binary and UQ + HESE settings (without TQ) apply term truncation by keeping the top α terms in each individual weight. In this case, μ is equal to α as the group size is 1 (*i.e.,* there is no grouping). We see that HESE substantially outperforms binary until the top 4 terms are kept per weight ($\mu = 4$) due to it requiring fewer terms. For the settings with TQ, TQ + binary and TQ + HESE, we use a group size of $g = 8$, with term budget α values of 8, 12, 16, 20, and 24 to generate comparable values of μ as in the settings without TQ. We find that TQ improves the performance of both the binary and HESE encoding methods, with TQ + HESE achieving the best performance.

D. Quantization Error Analysis

The reason for TQ's superior performance (*e.g.,* accuracy or perplexity) over UQ discussed in Section VI-A is due to TQ introducing less quantization error. Figure 18 shows the quantization error across the layers in ResNet-18 for 3 UQ

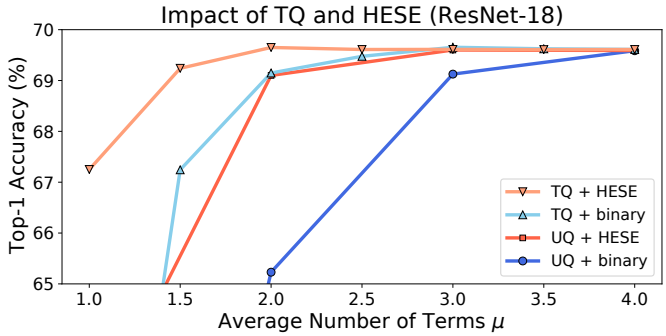

Fig. 17: Measuring the individual contributions of TQ and HESE in reducing number of terms while maintaining high classification accuracy.

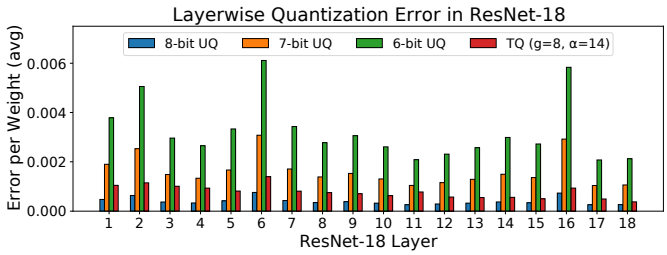

Fig. 18: The average quantization error (relative to the original 32-bit floating-point weights) across the convolutional layers in ResNet-18 for 3 UQ settings and one TQ setting.

settings (from 6-bit to 8-bit) and TQ with a group size $g = 8$ and a group budget $\alpha = 14$. Under this setting, the average number of terms μ per weight is less than 2. We see that TQ introduces a small amount of quantization error over 8-bit UQ, which makes sense as TQ is applied on top of 8-bit UQ. The 7-bit and 6-bit UQ settings truncate the low-order terms, leading to larger quantization error and reduced classification accuracy.

VII. Evaluation of FPGA Design of Term Quantization

In this section, we evaluate the hardware performance of an FPGA design of the TQ system described in Section V. We have synthesized our TQ system using Xilinx VC707 FPGA evaluation board. We first compare the performance of tMAC against a bit-parallel MAC (pMAC) in Section VII-A. Then, we demonstrate that our TQ system can be used to implement both UQ and TQ in Section VII-B. Finally, in Section VII-C, we compare our TQ system against other FPGA-based CNN accelerators on ResNet-18.

A. Comparing Performance of pMAC and tMAC

In this section, we evaluate the hardware performance of a single tMAC by comparing it against a bit-parallel MAC (pMAC) shown in Figure 10a. For both designs, we perform a group of MAC computation: $y_{out} = \sum_{i=1}^{g} x_i w_i + y_{in}$, where y_{in}, y_{out}, x_i and w_i are 32-bit, 32-bit, 8-bit and 8-bit, respectively, and $g = 8$ is the number of elements in the weight and data vectors (*i.e.,* the group size in TQ). In one cycle, the pMAC performs an 8-bit multiplication between x_i and w_i and a 32-bit accumulation between the result and y_{in}. Therefore, y_{out} is generated in $g = 8$ cycles. By comparison, the tMAC takes a variable number of cycles to process each multiplication in the group, depending on the number of term pairs in the multiplication. In total, it requires no more than $\alpha \times \beta$ cycles, where α is the group term budget and β is the maximum number of terms for each data value.

TABLE II: FPGA resource consumption of pMAC and tMAC.

	LUT	FF
pMAC	127	98
tMAC	30	32

TABLE III: Classification accuracy and energy efficiency comparison for the two MAC designs across four CNNs.

Model	MAC	α	β	g	Accuracy	Energy Eff.
Resnet-18	pMAC	-	-	-	69.62%	1.0×
	tMAC	12	3	8	69.60%	1.9×
VGG-16	pMAC	-	-	-	73.11%	1.0×
	tMAC	12	2	8	73.11%	2.8×
MoblNet-v2	pMAC	-	-	-	71.76%	1.0×
	tMAC	18	3	8	71.65%	1.4×
EffNet-b0	pMAC	-	-	-	75.99%	1.0×
	tMAC	16	3	8	75.84%	1.5×

Table II shows the FPGA resource consumption of the two MAC designs in terms of LookUp Tables (LUTs) and Flip-flops (FFs). The tMAC consumes 30 LUTs and 32 FFs, which is $4.2\times$ and $3.0\times$ less than the pMAC, which consumes 127 LUTs and 98 FFs. The tMAC requires less FPGA resources as it performs 3-bit exponent additions as opposed to 8-bit additions and 32-bit accumulation as in the pMAC.

We evaluate the two designs in terms of the energy efficiency, which is the ratio between the throughput and power consumption. Table III shows the energy efficiency and classification accuracy for the two designs across four CNNs. For the tMAC settings, different values of α and β are selected for each CNN such that the classification accuracy stays competitive with the baseline model (less than 0.15% difference in accuracy across all settings) while keeping the group size fixed ($g = 8$). For each CNN, the energy efficiency of both MAC designs is normalized to that of the pMAC. We observe that tMAC achieves superior energy efficiency compared to pMAC across the four CNNs. This reflects that pMAC needs to perform more work than tMAC, as our analysis in Section V-A shows.

B. System Comparison of UQ and TQ

In this section, we compare the hardware performance of TQ against UQ with the DNNs considered in Figure 15. The systolic array in the TQ system has 128 rows by 64 columns, with each systolic cell implementing a tMAC with a group size $g = 8$. The group budget α is chosen independently for

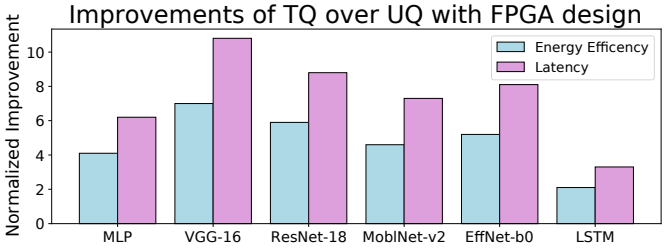

Fig. 19: Normalized energy efficiency and latency improvements of TQ over UQ. All models use a group size of $g = 8$. The group budget α is selected for each model such that it is within 0.15% accuracy of the corresponding UQ setting (α is 8, 12, 12, 18, 16, 20 for MLP, VGG-16, ResNet-18, MobileNet-V2, EfficientNet-b0, and LSTM, respectively). For data values, all models keep the top $\beta = 3$ terms except for VGG-16 which uses $\beta = 2$.

each network such that the TQ is within 0.15% accuracy of the UQ setting and 0.05 perplexity for the LSTM case. In this section, we use the same TQ system (Figure 9) for the implementation of both UQ and TQ in order to demonstrate the reconfigurability of our design. For the implementation of UQ, it does not require group-based ranking and HESE encoding, so we turn off these components of the hardware system to reduce dynamic power consumption. The control registers are configured based on Table I.

We evaluate our FPGA system with the following performance metrics: (1) *Average Processing latency* of the hardware system to generate the inference result, and (2) *Energy efficiency* or the average amount of energy required to process a single input sample. As shown in Figure 19, our TQ system outperforms the UQ across all models in terms of processing latency and energy efficiency. For more difficult tasks, such as Wikitext-2 for the LSTM, a more conservative group budget $\alpha = 20$ is selected, leading to less relative improvement over UQ. For overprovisioned models (*e.g.,* VGG-16), a more aggressive group budget of $\alpha = 8$ is used, leading to more substantial improvements in latency and energy efficiency.

C. Comparisons Against Other FPGA Designs

In this section, we evaluate our TQ system over ResNet-18 on ImageNet, using a group size $g = 8$ and group budget $\alpha = 16$. While an even larger group size could theoretically lead to additional savings, there are diminishing returns as shown in Figure 16 when comparing $g = 8$ and $g = 32$. Additionally, larger group sizes increase the complexity of the term comparator due to additional tree levels of A&C blocks.

We compare our TQ system with other FPGA-based accelerators which implement different CNN architectures (*e.g.,* AlexNet) on ImageNet. We evaluate these designs in terms of the average processing latency for the input samples, energy efficiency of the hardware system and classification accuracy. As shown in Table IV, our design achieves the highest classification accuracy (69.48%), energy efficiency (25.22 frames/J), and the second lowest latency $(7.21ms)$.

TABLE IV: Comparison of our FPGA implementation of ResNet-18 to other FPGA-based accelerators on ImageNet.

	[45]	[46]	[47]	[48]	Ours
FPGA Chip	VC706	Virtex-7	ZC706	ZC706	VC707
Acc. (%)	53.30%	55.70%	64.64%	N/A	69.48%
Frequency (MHz)	200	100	150	100	170
FF	51k(12%)	348k(40%)	127k(29%)	96k(22%)	316k(51%)
LUT	86k(39%)	236k(55%)	182k(83%)	148k(68%)	201k(65%)
DSP	808(90%)	3177(88%)	780(89%)	725(80%)	756(27%)
BRAM	303(56%)	1436(49%)	486(86%)	901(82%)	606(59%)
Latency (ms)	5.88	11.7	224	17.3	7.21
Energy eff. (frames/J)	23.6	8.39	0.46	6.13	25.22

Our hardware system achieves improved performance for multiple reasons. First, TQ coupled with the proposed HESE encoding greatly reduces the amount of term-pair multiplications, resulting in a reduced number of cycles in tMACs. TQ allows tMACs to achieve a much tighter processing bound of $3 \times k$ pairs per group as opposed to $7 \times 7 \times g$ in the case of standard binary encoding without TQ. Second, the bit-serial design of the coefficient accumulator in tMAC together with the systolic architecture of the computing engine leads to a highly regular layout with low routing complexity.

VIII. Conclusion

We proposed term quantization (TQ) as a general run-time approach for furthering quantization on already quantized DNNs. Departing from conventional quantization that operates on individual values, TQ is a group-based method that keeps a fixed number of largest terms within a group of values. TQ leverages the weight and data distributions of DNNs, so that it can achieve good model performance even with a small group budget. We measure the computation cost of TQ using the number of term-pair multiplications per inference sample. Under this clearly defined cost proxy, we have shown that TQ significantly lowers computation costs for MLPs, CNNs, and LSTMs. As shown in Section VII-B, this reduction in operations translates to improved energy efficiency and reduced latency over conventional quantization for our FPGA system. Furthermore, our FPGA system demonstrates that by changing a small number of control bits we can reconfigure a quantized computation under conventional quantization to one under TQ, and vice versa (Table I). As a follow-on work, we could train models with TQ-aware training, so that post-training TQ based on such pre-trained models could allow further performance improvement.

Quantization is one of most widely used approaches in streamlining DNNs; TQ proposed in this paper brings the success of the quantization approach to another level.

IX. Acknowledgements

This work is supported in part by the Air Force Research Laboratory under agreement number FA8750-18-1-0112, a gift from MediaTek USA and a Joint Development Project with TSMC.

REFERENCES

[1] "Smart compose: Using neural networks to help write emails." Available at: "https://ai.googleblog.com/2018/05/smart-compose-using-neural-networks-to.html.

[2] D. Lin, S. Talathi, and S. Annapureddy, "Fixed point quantization of deep convolutional networks," in *International Conference on Machine Learning*, pp. 2849–2858, 2016.

[3] A. D. Booth, "A signed binary multiplication technique," *The Quarterly Journal of Mechanics and Applied Mathematics*, vol. 4, no. 2, pp. 236–240, 1951.

[4] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," *arXiv preprint arXiv:1510.00149*, 2015.

[5] W. Wen, C. Wu, Y. Wang, Y. Chen, and H. Li, "Learning structured sparsity in deep neural networks," in *Advances in Neural Information Processing Systems*, pp. 2074–2082, 2016.

[6] G. Huang, S. Liu, L. van der Maaten, and K. Q. Weinberger, "Condensenet: An efficient densenet using learned group convolutions," *arXiv preprint arXiv:1711.09224*, 2017.

[7] Y. He, X. Zhang, and J. Sun, "Channel pruning for accelerating very deep neural networks," in *International Conference on Computer Vision (ICCV)*, vol. 2, 2017.

[8] J.-H. Luo, J. Wu, and W. Lin, "Thinet: A filter level pruning method for deep neural network compression," *arXiv preprint arXiv:1707.06342*, 2017.

[9] S. Narang, E. Undersander, and G. F. Diamos, "Block-sparse recurrent neural networks," *CoRR*, vol. abs/1711.02782, 2017.

[10] S. Gray, A. Radford, and D. Kingma, "Gpu kernels for block-sparse weights." https://s3-us-west-2.amazonaws.com/openai-assets/blocksparse/blocksparsepaper.pdf, 2017. [Online; accessed 12-January-2018].

[11] H. T. Kung, B. McDanel, and S. Q. Zhang, "Packing sparse convolutional neural networks for efficient systolic array implementations: Column combining under joint optimization," *24th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019.

[12] A. Ren, T. Zhang, S. Ye, J. Li, W. Xu, X. Qian, X. Lin, and Y. Wang, "Admm-nn: An algorithm-hardware co-design framework of dnns using alternating direction methods of multipliers," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 925–938, ACM, 2019.

[13] M. Courbariaux, Y. Bengio, and J.-P. David, "Training deep neural networks with low precision multiplications," *arXiv preprint arXiv:1412.7024*, 2014.

[14] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, "Deep learning with limited numerical precision," in *International Conference on Machine Learning*, pp. 1737–1746, 2015.

[15] C. Zhu, S. Han, H. Mao, and W. J. Dally, "Trained ternary quantization," *arXiv preprint arXiv:1612.01064*, 2016.

[16] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, "Quantized neural networks: Training neural networks with low precision weights and activations," *The Journal of Machine Learning Research*, vol. 18, no. 1, pp. 6869–6898, 2017.

[17] E. Park, J. Ahn, and S. Yoo, "Weighted-entropy-based quantization for deep neural networks," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 5456–5464, 2017.

[18] S. Kapur, A. Mishra, and D. Marr, "Low precision rnns: Quantizing rnns without losing accuracy," *arXiv preprint arXiv:1710.07706*, 2017.

[19] A. Zhou, A. Yao, Y. Guo, L. Xu, and Y. Chen, "Incremental network quantization: Towards lossless cnns with low-precision weights," *arXiv preprint arXiv:1702.03044*, 2017.

[20] P. Wang and J. Cheng, "Fixed-point factorized networks," in *Computer Vision and Pattern Recognition (CVPR), 2017 IEEE Conference on*, pp. 3966–3974, IEEE, 2017.

[21] C.-Y. Chen, J. Choi, K. Gopalakrishnan, V. Srinivasan, and S. Venkataramani, "Exploiting approximate computing for deep learning acceleration," in *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 821–826, IEEE, 2018.

[22] E. Park, D. Kim, and S. Yoo, "Energy-efficient neural network accelerator based on outlier-aware low-precision computation," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pp. 688–698, IEEE, 2018.

[23] E. Park, S. Yoo, and P. Vajda, "Value-aware quantization for training and inference of neural networks," in *Proceedings of the European Conference on Computer Vision (ECCV)*, pp. 580–595, 2018.

[24] Q. Hu, P. Wang, and J. Cheng, "From hashing to cnns: Training binaryweight networks via hashing," *arXiv preprint arXiv:1802.02733*, 2018.

[25] B. McDanel, S. Q. Zhang, H. T. Kung, and X. Dong, "Full-stack optimization for accelerating cnns using powers-of-two weights with fpga validation," *International Conference on Supercomputing*, 2019.

[26] A. Li, T. Geng, T. Wang, M. Herbordt, S. L. Song, and K. Barker, "Bstc: a novel binarized-soft-tensor-core design for accelerating bit-based approximated neural nets," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–30, 2019.

[27] M. Courbariaux, Y. Bengio, and J.-P. David, "Binaryconnect: Training deep neural networks with binary weights during propagations," in *Advances in neural information processing systems*, pp. 3123–3131, 2015.

[28] J. Albericio, A. Delmás, P. Judd, S. Sharify, G. O'Leary, R. Genov, and A. Moshovos, "Bit-pragmatic deep neural network computing," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 382–394, ACM, 2017.

[29] A. Delmas, P. Judd, D. M. Stuart, Z. Poulos, M. Mahmoud, S. Sharify, M. Nikolic, and A. Moshovos, "Bit-tactical: Exploiting ineffectual computations in convolutional neural networks: Which, why, and how," *24th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019.

[30] H. T. Kung, "Why systolic architectures?," *IEEE Computer*, vol. 15, pp. 37–46, 1982.

[31] C. Zhang, S. Bengio, M. Hardt, B. Recht, and O. Vinyals, "Understanding deep learning requires rethinking generalization," *arXiv preprint arXiv:1611.03530*, 2016.

[32] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," *arXiv preprint arXiv:1502.03167*, 2015.

[33] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.

[34] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*, pp. 248–255, IEEE, 2009.

[35] A. Avizienis, "Signed-digit numbe representations for fast parallel arithmetic," *IRE Transactions on electronic computers*, no. 3, pp. 389–400, 1961.

[36] B. L. Drake, R. P. Bocker, M. E. Lasher, R. H. Patterson, and W. J. Miceli, "Photonic computing using the modified signed-digit number representation," *Optical Engineering*, vol. 25, no. 1, p. 250138, 1986.

[37] J. Jedwab and C. J. Mitchell, "Minimum weight modified signed-digit representations and fast exponentiation," *Electronics Letters*, vol. 25, no. 17, pp. 1171–1172, 1989.

[38] Y. LeCun, "The mnist database of handwritten digits," *http://yann. lecun. com/exdb/mnist/*, 1998.

[39] K. Simonyan, A. Vedaldi, and A. Zisserman, "Deep inside convolutional networks: Visualising image classification models and saliency maps," *arXiv preprint arXiv:1312.6034*, 2013.

[40] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "Mobilenetv2: Inverted residuals and linear bottlenecks," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 4510–4520, 2018.

[41] M. Tan and Q. V. Le, "Efficientnet: Rethinking model scaling for convolutional neural networks," *arXiv preprint arXiv:1905.11946*, 2019.

[42] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[43] S. Merity, C. Xiong, J. Bradbury, and R. Socher, "Pointer sentinel mixture models," *arXiv preprint arXiv:1609.07843*, 2016.

[44] J. H. Lee, S. Ha, S. Choi, W.-J. Lee, and S. Lee, "Quantization for rapid deployment of deep neural networks," *arXiv preprint arXiv:1810.05488*, 2018.

[45] X. Zhang, J. Wang, C. Zhu, Y. Lin, J. Xiong, W.-m. Hwu, and D. Chen, "Dnnbuilder: an automated tool for building high-performance dnn hardware accelerators for fpgas," in *Proceedings of the International Conference on Computer-Aided Design*, p. 56, ACM, 2018.

[46] Y. Shen, M. Ferdman, and P. Milder, "Maximizing cnn accelerator efficiency through resource partitioning," in *Computer Architecture (ISCA), 2017 ACM/IEEE 44th Annual International Symposium on*, pp. 535–547, IEEE, 2017.

[47] J. Qiu, J. Wang, S. Yao, K. Guo, B. Li, E. Zhou, J. Yu, T. Tang, N. Xu, S. Song, *et al.*, "Going deeper with embedded fpga platform for convolutional neural network," in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 26–35, ACM, 2016.

[48] Q. Xiao, Y. Liang, L. Lu, S. Yan, and Y.-W. Tai, "Exploring heterogeneous algorithms for accelerating deep convolutional neural networks on fpgas," in *Proceedings of the 54th Annual Design Automation Conference 2017*, p. 62, ACM, 2017.