UCNN: Exploiting Computational Reuse in Deep Neural Networks via Weight Repetition

Kartik Hegde, Jiyong Yu, Rohit Agrawal, Mengjia Yan, Michael Pellauer*, Christopher W. Fletcher

University of Illinois at Urbana-Champaign

*NVIDIA

{kvhegde2, jiyongy2, rohita2, myan8}@illinois.edu, mpellauer@nvidia.com, cwfletch@illinois.edu

Abstract—Convolutional Neural Networks (CNNs) have begun to permeate all corners of electronic society (from voice recognition to scene generation) due to their high accuracy and machine efficiency per operation. At their core, CNN computations are made up of multi-dimensional dot products between weight and input vectors. This paper studies how *weight repetition*—when the same weight occurs multiple times in or across weight vectors—can be exploited to save energy and improve performance during CNN inference. This generalizes a popular line of work to improve efficiency from CNN weight sparsity, as reducing computation due to repeated zero weights is a special case of reducing computation due to repeated weights.

To exploit weight repetition, this paper proposes a new CNN accelerator called the Unique Weight CNN Accelerator (UCNN). UCNN uses weight repetition to reuse CNN sub-computations (e.g., dot products) and to reduce CNN model size when stored in off-chip DRAM—both of which save energy. UCNN further improves performance by exploiting sparsity in weights. We evaluate UCNN with an accelerator-level cycle and energy model and with an RTL implementation of the UCNN processing element. On three contemporary CNNs, UCNN improves throughput-normalized energy consumption by $1.2\times \sim 4\times$, relative to a similarly provisioned baseline accelerator that uses Eyeriss-style sparsity optimizations. At the same time, the UCNN processing element adds only 17-24% area overhead relative to the same baseline.

I. INTRODUCTION

We are witnessing an explosion in the use of Deep Neural Networks (DNNs), with major impacts on the world's economic and social activity. At present, there is abundant evidence of DNN's effectiveness in areas such as classification, vision, and speech [1], [2], [3], [4], [5]. Of particular interest are Convolutional Neural Networks (CNNs), which achieve state-of-the-art performance in many of these areas, such as image/temporal action recognition [6], [7] and scene generation [8]. An ongoing challenge is to bring CNN *inference*—where the CNN is deployed in the field and asked to answer online queries—to edge devices, which has inspired CNN architectures ranging from CPUs to GPUs to custom accelerators [9], [10], [11], [12]. A major challenge along this line is that CNNs are notoriously compute intensive [9], [13]. It is imperative to find new ways to reduce the work needed to perform inference.

At their core, CNN computations are parallel dot products. Consider a 1-dimensional convolution (i.e., a simplified CNN kernel), which has filter $\{a, b, a\}$ and input $\{x, y, z, k, l, \dots\}$. We refer to elements in the input as the activations, elements in the filter as the weights and the number of weights in the filter as the filter's size (3 in this case). The output is
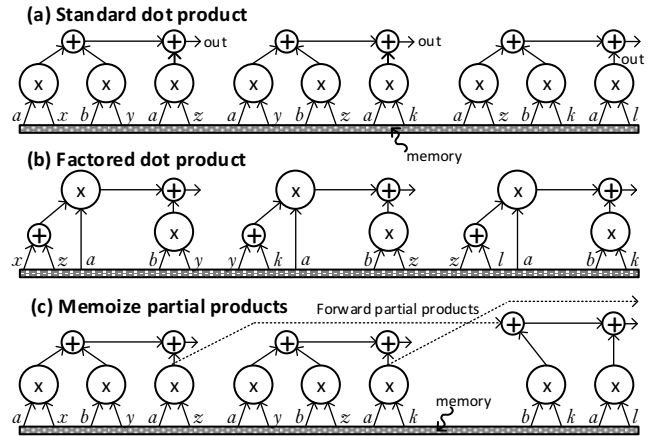

Fig. 1. Standard (a) and different optimized (b, c) 1D convolutions that take advantage of repeated weight a. Arrows out of the grey bars indicate input/filter memory reads. Our goal is to reduce memory reads, multiplications and additions while obtaining the same result.

computed by sliding the filter across the input and taking a filter-sized dot product at each position (i.e., $\{ax + by + az, ay + bz + ak, \dots\}$), as shown in Figure 1a. When evaluated on hardware, this computation entails reading each input and weight from memory, and performing a multiply-accumulate (MAC) on that input-weight pair. In this case, each output performs 6 memory reads (3 weights and 3 inputs), 3 multiplies and 2 adds.

In this paper, we explore how CNN hardware accelerators can eliminate superfluous computation by taking advantage of *repeated weights*. In the above example, we have several such opportunities because the weight a appears twice. First (Figure 1b), we can factor dot products as sum-of-products-of-sums expressions, saving 33% multiplies and 16% memory reads. Second (Figure 1c), each partial product $a * \text{input}$ computed at the filter's right-most position can be memoized and re-used when the filter slides right by two positions, saving 33% multiples and memory reads. Additional opportunities are explained in Section III. Our architecture is built on these two ideas: factorization and memoization, both of which are only possible given repeated weights (two a's in this case).

Reducing computation via weight repetition is possible due to CNN filter design/weight quantization techniques, and is inspired by recent work on sparse CNN accelerators. A filter is guaranteed to have repeated weights when the filter size exceeds

IEEE
computer
society

the number of unique weights, due to the pigeonhole principle. Thus, out-of-the-box (i.e., not re-trained [14]) networks may see weight repetition already. For example, representing each weight in 8 bits [13] implies there are ≤ 256 unique weights, whereas filter size can be in the thousands of weights [6], [15], [16]. Augmenting this further is a rich line of work to *quantize weights* [14], [17], [18], which strives to decrease the number of unique weights without losing significant classification accuracy. For example, INQ [18] and TTQ [17] use 17 and 3 unique weights, respectively, without changing filter size. Finally, innovations [10], [19], [12] that exploit CNN sparsity (zero weights/activations) inspire and complement weight repetition. Weight repetition generalizes this optimization: reducing computation due to repeated zero weights is a special case of reducing computation due to repeated weights.

Exploiting weight repetition while getting a net efficiency win, however, is challenging for two reasons. First, as with sparse architectures, tracking repetition patterns is difficult because they are irregular. Second, naïve representations of tracking metadata require a large amount of storage. This is a serious problem due to added system energy cost of transporting metadata throughout the system (e.g., reading the model from DRAM, PCI-e, etc).

This paper addresses these challenges with a novel CNN accelerator architecture called UCNN, for Unique Weight CNN Accelerator. UCNN is based on two main ideas. First, we propose a *factorized dot product dataflow* which reduces multiplies and weight memory reads via weight repetition, and improves performance via weight sparsity [19], [12]. Second, we propose *activation group reuse*, which builds on dot product factorization to reduce input memory reads, weight memory reads, multiplies and adds per dot product, while simultaneously compressing the CNN model size. The compression rate is competitive to that given by aggressive weight quantization schemes [18], [17], and gives an added ability to exploit weight repetition. We employ additional architectural techniques to amortize the energy cost of irregular accesses and to reduce hardware area overhead.

Contributions. To summarize, this paper makes the following contributions.

1) We introduce new techniques—including dot product factorization and activation group reuse—to improve CNN efficiency by exploiting weight repetition in and across CNN filters.
2) We design and implement a novel CNN accelerator, called UCNN, that improves performance and efficiency per dot product by using the aforementioned techniques.
3) We evaluate UCNN using an accelerator-level cycle and energy model as well as an RTL prototype of the UCNN processing element. On three contemporary CNNs, UCNN improves throughput-normalized energy consumption by $1.2\times \sim 4\times$, relative to a similarly provisioned baseline accelerator that uses Eyeriss-style sparsity optimizations. At the same time, the UCNN processing element adds only 17-24% area overhead relative to the same baseline.

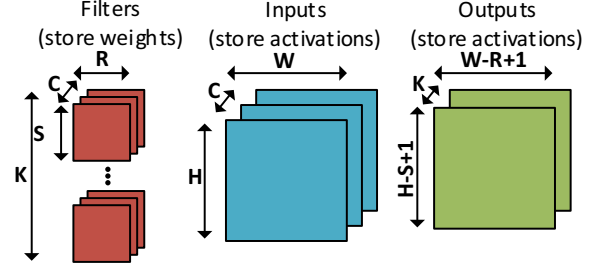We note that while our focus is to accelerate CNNs due

Fig. 2. CNN parameters per convolutional layer.

to their central role in many problems, weight repetition is a general phenomena that can be exploited by any DNN based on dot products, e.g., multilayer perceptions. Further, some of our techniques, e.g., dot product factorization, work out of the box for non-CNN algorithms.

Paper outline. The rest of the paper is organized as follows. Section II gives background on CNNs and where weight repetition occurs in modern networks. Section III presents strategies for CNN accelerators to reduce work via weight repetition. Section IV proposes a detailed processing element (PE)-level architecture to improve efficiency via weight repetition. Section V gives a dataflow and macro-architecture for the PE. Section VI evaluates our architecture relative to dense baselines. Section VII covers related work. Finally Section VIII concludes.

II. BACKGROUND

A. CNN Background

CNNs are made up of multiple layers, where the predominant layer type is a multi-dimensional convolution. Each convolutional layer involves a 3-dimensional ($W \times H \times C$) input and K 3-dimensional ($R \times S \times C$) filters. Convolutions between the filters and input form a 3-dimensional ($W - R + 1) \times (H - S + 1) \times K$ output. These parameters are visualized in Figure 2. C and K denote the layer's input and output channel count, respectively. We will omit '$\times$' from dimensions for brevity when possible, e.g., $W \times H \times C \rightarrow WHC$.

CNN inference. As with prior work [12], [19], [10], this paper focuses on CNN inference, which is the online portion of the CNN computation run on, e.g., edge devices. The inference operation for convolutional layers (not showing bias terms, and for unit stride) is given by

$$O[(k,x,y)] = \sum_{c=0}^{C-1}\sum_{r=0}^{R-1}\sum_{s=0}^{S-1} F[(k,c,r,s)] * I[(c,x+r,y+s)] \quad (1)$$
$$0 \leq k < K, 0 \leq x < W - R + 1, 0 \leq y < H - S + 1$$

where O, I and F are outputs (activations), inputs (activations) and filters (weights), respectively. Outputs become inputs to the next layer. Looking up O, I and F with a tuple is notation for a multi-dimensional array lookup. As is the case with other works targeting inference [11], we assume a batch size of one.

We remark that CNNs have several other layer types including non-linear scaling [23] layers, down-sampling/pooling
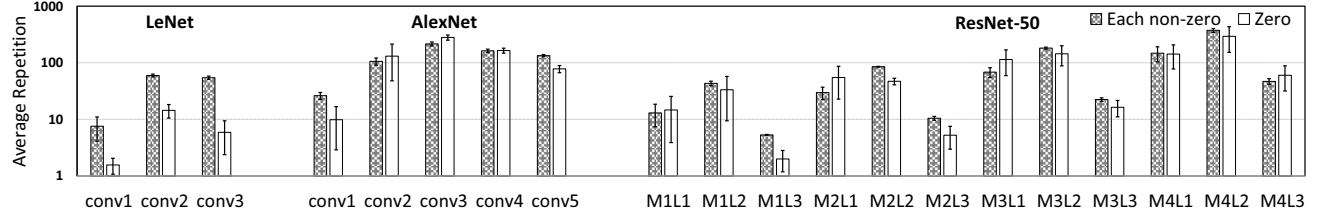
675

Fig. 3. Weight repetition per filter, averaged across all filters, for select layers in a Lenet-like CNN [20], AlexNet [6] and ResNet-50 [16]. All networks are trained with INQ [18]. LeNet was trained on CIFAR-10 [21] and AlexNet/ResNet were trained on ImageNet [22]. MxLy stands for "module x, layer y." In the case of ResNet, we show one instance of each module, where repetition is averaged across filters in the layer. Note that the error bars represent the standard deviation of weight repetition in each layer.

layers and fully connected layers. We focus on accelerating convolutional layers as they constitute the majority of the computation [24], but explain how to support other layer types in Section IV-E.

B. Weight Repetition in Modern CNNs

We make a key observation that, while CNN filter dimensions have been relatively constant over time, the number of unique weights in each filter has decreased dramatically. This is largely due to several successful approaches to compress CNN model size [25], [26], [14], [18], [17]. There have been two main trends, both referred to as *weight quantization* schemes. First, to decrease weight numerical precision, which reduces model size and the cost of arithmetic [13]. Second, to use a small set of high-precision weights [14], [18], [17], which also reduces model size but can enable higher accuracy than simply reducing precision.

Many commercial CNNs today are trained with reduced, e.g., 8 bit [13], precision per weight. We refer to the number of unique weights in the network as U. Thus, with 8 bit weights $U \leq 2^8 = 256$. Clearly, weight repetition within and across filters is guaranteed as long as $U < R * S * C$ and $U < R * S * C * K$, respectively. This condition is common in contemporary CNNs, leading to a guaranteed weight repetition in modern networks. For example, every layer except the first layer in ResNet-50 [16] has more than 256 weights per filter and between $K = 64$ to $K = 512$ filters.

A complementary line of work shows it is possible to more dramatically reduce the number of unique weights, while maintaining state-of-the-art accuracy, by decoupling the number of unique weights from the numerical precision per weight [14], [18], [17]. Figure 3 shows weight repetition for several modern CNNs trained with a scheme called Incremental Network Quantization (INQ) [18]. INQ constrains the trained model to have only $U = 17$ unique weights (16 non-zero weights plus zero) and achieves state-of-the-art accuracy on many contemporary CNNs. Case in point, Figure 3 shows a LeNet-like CNN from Caffe [20] trained on CIFAR-10 [21], and AlexNet [6] plus ResNet-50 [16] trained on ImageNet [22], which achieved 80.16%, 57.39% and 74.81% top-1 accuracy, respectively.

Figure 3 shows that weight repetition is widespread and abundant across a range of networks of various sizes and depths.

TABLE I
UCNN PARAMETERS.

Name	Description	Defined
U	Number of unique weights per CNN layer	II
iiT	Indirection table into input buffer	III-A
wiT	Indirection table into filter buffer	III-A
G	Number of filters grouped for act. group reuse	III-B

We emphasize that repetition counts for the non-zero column in Figure 3 are the average repetition for *each* non-zero weight value *within each* filter. We see that each non-zero weight is seldom repeated less than 10 times. Interestingly, the repetition count per non-zero is similar to that of the zero weight for most layers. This implies that the combined repetitions of non-zero weights (as there are $U - 1$ non-zero weights) can dwarf the repetitions of zero weights. The key takeaway message is that there is a large un-tapped potential opportunity to exploit repetitions in non-zero weights.

III. EXPLOITING WEIGHT REPETITION

We now discuss opportunities that leverage weight repetition to reduce work (save energy and cycles), based on refactoring and reusing CNN sub-computations. From Section II, there are K CNN filters per layer, each of which spans the three dimensions of *RSC*. Recall, R and S denote the filter's spatial dimensions and C denotes the filter's channels.

We first present *dot product factorization* (Section III-A), which saves multiplies by leveraging repeated weights within a single filter, i.e., throughout the *RSC* dimensions. We then present a generalization of dot product factorization, called *activation group reuse* (Section III-B), to exploit repetition within and across filters, i.e., throughout the *RSCK* dimensions. Lastly, we remark on a third type of reuse that we do not exploit in this paper (Section III-C) but may be of independent interest.

For clarity, we have consolidated parameters and terminology unique to this paper in Table I.

A. Dot Product Factorization

Given each dot product in the CNN (an *RSC*-shape filter MACed to an *RSC* sub-region of input), our goal is to reduce the number of multiplies needed to evaluate that dot product. This can be accomplished by factorizing out common weights

676

in the dot product, as shown in Figure 1b. That is, input activations that will be multiplied with the same weight (e.g., x, z and y, k and z, l in Figure 1b) are grouped and summed locally, and only that sum is multiplied to the repeated weight. We refer to groups of activations summed locally as *activation groups* — we use this term extensively in the rest of the paper. To summarize:

1) Each activation group corresponds to one unique weight in the given filter.
2) The total number of activation groups per filter is equal to the number of unique weights in that filter.
3) The size of each activation group is equal to the repetition count for that group's weight in that filter.

We can now express dot product factorization by rewriting the Equation 1 as

$$O[(k,x,y)] = \sum_{i=0}^{U} \left(F[\text{wiT}[(k,i)]] * \sum_{j=0}^{\text{gsz}(k,i)-1} I[\text{iiT}[(k,i,j)]] \right) \quad (2)$$

O, F and I are outputs, filters and inputs from Equation 1, $\text{gsz}(k,i)$ indicates the size of the i-th activation group for the k-th filter and U represents the number of unique weights in the network (or network layer). Note that each filter can have a different number of unique weights due to an irregular weight distribution. That is, some activation groups may be "empty" for a given filter. For simplicity, we assume each filter has U activation groups in this section, and handle the empty corner case in Section IV-C.

Activation groups are spread out irregularly within each *RSC* sub-region of input. Thus, we need an indirection table to map the locations of each activation that corresponds to the same unique weight. We call this an *input indirection table*, referred to as iiT. The table iiT reads out activations from the input space in activation group-order. That is, $\text{iiT}[(k,i,0)]\ldots\text{iiT}[(k,i,\text{gsz}(k,i)-1)]$ represents the indices in the input space corresponding to activations in the i-th activation group for filter k.

Correspondingly, we also need to determine which unique weight should be multiplied to each activation group. We store this information in a separate *weight indirection table*, referred to as wiT. $\text{wiT}[(k,i)]$ points to the unique weight that must be multiplied to the i-th activation group for filter k. We emphasize that, since the weights are static for a given model, both of the above tables can be pre-generated offline.

Savings. The primary benefit from factorizing the dot product is reduced multiplications per dot product. Through the above scheme, the number of multiplies per filter reduces to the number of unique weights in the filter (e.g., 17 for INQ [18]), regardless of the size of the filter or activation group. Referring back to Figure 3, average multiplication savings would be the height of each bar, and this ranges from $5\times$ to $373\times$. As discussed in Section II, even out-of-the-box networks are guaranteed to see savings.

An important special case is the zero weight, or when $F[\text{wiT}[(k,i)]] = 0$. Then, the inner loop to sum the activation group and the associated multiplication is skipped.
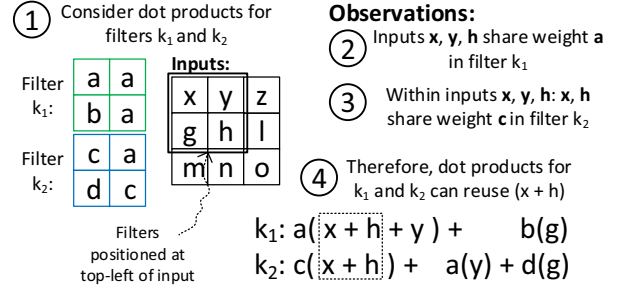

Fig. 4. Activation group reuse example ($G = 2$).

Costs. The multiply and sparsity savings come at the cost of storage overhead for the input and weight indirection tables, which naïvely are the size of the original dense weights, as well as the energy costs to lookup inputs/weights through these tables. We introduce several techniques to reduce the size of each of these tables (Sections IV-B to IV-C). We also amortize the cost of looking up these tables through a novel vectorization scheme (Section IV-D).

B. Activation Group Reuse

Dot product factorization reduces multiplications by exploiting weight repetition within the filter. We can generalize the idea to simultaneously exploit repetitions across filters, using a scheme called *activation group reuse*. The key idea is to exploit the overlap between two or more filters' activation groups. In Figure 4, activations $x+h+y$ form an activation group for filter k_1. Within this activation group, filter k_2 has a *sub-activation group* which is $x+h$. The intersection of these two ($x+h$) can be reused across the two filters.

Formally, we can build the sub-activation groups for filter k_2, within filter k_1's i-th activation group, as follows. First, we build the activation group for k_1:

$$A(k_1,i) = \{\text{iiT}[(k_1,i,j)] : j \in [0, \text{gsz}(k_1,i))\}$$

Then, we build up to U sub-activation groups for k_2 by taking set intersections. That is, for $i' = 0, \ldots, U-1$, the i'-th sub-activation group for k_2 is given by:

$$A(k_1,i) \bigcap A(k_2,i')$$

We can generalize the scheme to find overlaps across G filters. When $G = 1$, we have vanilla dot product factorization (Section III-A). The discussion above is for $G = 2$. When $G > 2$, we recursively form set intersections between filters k_g and k_{g+1}, for $g = 1, \ldots, G-1$. That is, once sub-activation groups for a filter k_2 are formed, we look for "sub-sub" activation groups within a filter k_3 which fall within the sub-groups for k_2, etc. Formally, suppose we have a gth-level activation group T_g for filter k_g. To find the $(g+1)$th-level activation groups for filter k_{g+1} within T_g, we calculate $T_g \bigcap A(k_{g+1},i')$ for $i' = 0, \ldots, U-1$, which is analogous to how intersections were formed for the $G = 2$ case.

As mentioned previously, irregular weight distributions may mean that there are less than U unique weights in filter k_{g+1}

that overlap with a given gth-level activation group for filter k_g. We discuss how to handle this in Section IV-C.

Savings. Activation group reuse can bring significant improvements in two ways:

1) *Reduced input buffer reads and arithmetic operations*: From Figure 4, we can eliminate the buffer reads and additions for reused sub-expressions like $x + h$. The scheme simultaneously saves multiplies as done in vanilla dot product factorization.

2) *Compressed input indirection table* iiT: Since we do not need to re-read the sub-, sub-sub-, etc. activation groups for filters $k_2, \ldots, k_G$, we can reduce the size of the input indirection table iiT by an $O(G)$ factor. We discuss this in detail in Section IV-C.

How prevalent is Activation Group Reuse? Activation group reuse is only possible when there are overlaps between the activation groups of two or more filters. If there are no overlaps, we cannot form compound sub-activation group expressions that can be reused across the filters. These overlaps are likely to occur when the filter size $R * S * C$ is larger than U^G, i.e., the number of unique weights to the G-th power. For example, for $(R, S, C) = (3, 3, 256)$ and $U = 8$, we expect to see overlaps between filter groups up to size $G = 3$ filters.

We experimentally found that networks retrained with INQ [18] ($U = 17$) and TTQ [17] ($U = 3$) can enable $G > 1$. In particular, INQ satisfies between $G = 2$ to 3 and TTQ satisfies $G = 6$ to 7 for a majority of ResNet-50 layers. Note that these schemes can simultaneously achieve competitive classification accuracy relative to large U schemes.

C. Partial Product Reuse

We make the following additional observation. While dot product factorization looks for repetitions in each *RSC*-dimensional filter, it is also possible to exploit repetitions across filters, within the same input channel. That is, across the *RSK* dimensions for each input channel. This idea is shown for 1D convolution in Figure 1c. In CNNs, for each input channel C, if $w = F[(k_1, c, r_1, s_1)] = F[(k_2, c, r_2, s_2)]$ and $(k_1, r_1, s_1) \neq (k_2, r_2, s_2)$, partial products formed with weight w can be reused across the filters, for the same spatial position, and as the filters slide. We do not exploit this form of computation reuse further in this paper, as it is not directly compatible with the prior two techniques.

IV. PROCESSING ELEMENT ARCHITECTURE

In this section, we will describe Processing Element (PE) architecture, which is the basic computational unit in the accelerator. We will first describe the PE of an efficient Dense CNN accelerator, called DCNN. We will then make PE-level changes to the DCNN design to exploit the weight repetition-based optimizations from Section III. This is intended to give a clear overview of how the UCNN design evolves over an efficient dense architecture and also to form a baseline for evaluations in Section VI.

The overall accelerator is made up of multiple PEs and a global buffer as depicted in Figure 5. The global buffer is responsible for scheduling work to the PEs. We note that aside from changes to the PEs, the DCNN and UCNN accelerators (including their dataflow [27]) are essentially the same. We provide details on the overall (non-PE) architecture and dataflow in Section V.
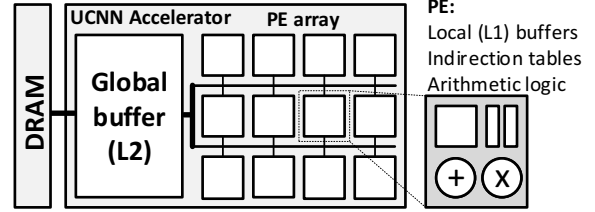

Fig. 5. Chip-level DCNN/UCNN architecture. Indirection tables are UCNN only.

A. Baseline Design: DCNN PE

The DCNN and UCNN PE's unit of work is to compute a dot product between an *RSC* region of inputs and one or more filters. Recall that each dot product corresponds to all three loops in Equation 1, for a given (k, x, y) tuple.

To accomplish this task, the PE is made up of an input buffer, weight buffer, partial sum buffer, control logic and MAC unit (the non-grey components in Figure 6). At any point in time, the PE works on a filter region of size RSC_t where $C_t \leq C$, i.e., the filter is tiled in the channel dimension. Once an RSC_t region is processed, the PE will be given the next RSC_t region until the whole RSC-sized dot product is complete.

Since this is a dense CNN PE, its operation is fairly straightforward. Every element of the filter is element-wise multiplied to every input element in the corresponding region, and the results are accumulated to provide a single partial sum. The partial sum is stored in the local partial sum buffer and is later accumulated with results of the dot products over the next RSC_t-size filter tile.

Datapath. The datapath is made up of a fixed point multiplier and adder as shown in Figure 6 ①. Once the data is available in the input and weight buffers, the control unit feeds the datapath with a weight and input element every cycle. They are MACed into a register that stores a partial sum over the convolution operation before writing back to the partial sum buffer. Together, we refer to this scalar datapath as a *DCNN lane*.

Vectorization. There are multiple strategies to vectorize this PE. For example, we can vectorize across output channels (amortizing input buffer reads) by replicating the lane and growing the weight buffer capacity and output bus width. DCNN and UCNN will favor different vectorization strategies, and we specify strategies for each later in the section and in the evaluation.

B. Dot Product Factorization

We now describe how to modify the DCNN architecture to exploit dot product factorization (Section III-A). The UCNN PE design retains the basic design and components of the DCNN
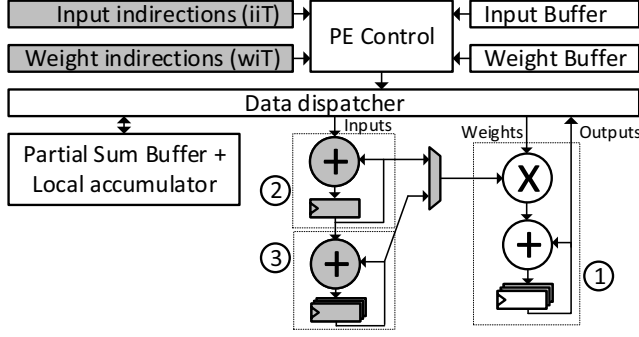
678

Fig. 6. DCNN/UCNN PE Architecture. Every component in grey is addition over the DCNN PE to design the UCNN PE. ① represents a DCNN vector lane and ①-③ represents a UCNN vector lane. ② is an accumulator added to sum activation groups for dot product factorization. ③ is an additional set of accumulators for storing sub-activation group partial sums. There are G and $G-1$ accumulator registers in components ① and ③, respectively.

PE along with its dataflow (Section IV-A). As described by Equation 2, now the dot product operation is broken down into two separate steps in hardware:

1) An inner loop which sums all activations within an activation group.
2) An outer loop which multiplies the sum from Step 1 with the associated weight and accumulates the result into the register storing the partial sum.

Indirection table sorting. Compared to DCNN, we now additionally need two indirection tables: the input indirection table (iiT) and the weight indirection table (wiT) as discussed in Section III-A. Since we work on an RSC_t-size tile at a time, we need to load RSC_t entries from both indirection tables into the PE at a time. Following Equation 2 directly, each entry in iiT and wiT is a $\lceil \log_2 RSC_t \rceil$ and $\lceil \log_2 U \rceil$-bit pointer, respectively.

To reduce the size of these indirection tables and to simplify the datapath, we sort entries in the input and weight indirection tables such that reading the input indirections sequentially looks up the input buffer in activation group-order. The weight indirection table is read in the same order. Note that because sorting is a function of weight repetitions, it can be performed offline.

Importantly, the sorted order implies that each weight in the weight buffer need only be read out once per activation group, and that the weight indirection table can be implemented as a single bit per entry (called the *group transition* bit), to indicate the completion of an activation group. Specifically, the next entry in the weight buffer is read whenever the group transition bit is set and the weight buffer need only store U entries.

As mentioned in Section III-A, we don't store indirection table entries that are associated with the zero weight. To skip zeros, we sort the zero weight to the last position and encode a "filter done" message in the existing table bits when we make the group transition to zero. This lets UCNN skip zero weights as proposed by previous works [19], [12] and makes

the exploitation of weight sparsity a special case of weight repetition.

Datapath. The pipeline follows the two steps from the beginning of the section, and requires another accumulator to store the activation group sum as reflected in Figure 6 ②. As described above, the sorted input and weight indirection tables are read sequentially. During each cycle in Step 1, the input buffer is looked up based on the current input indirection table entry, and summed in accumulator ② until a group transition bit is encountered in the weight indirection table. In Step 2, the next weight from the weight buffer is multiplied to the sum in the MAC unit (Figure 6 ①). After every activation group, the pipeline performs a similar procedure using the next element from the weight buffer.

Arithmetic bitwidth. This design performs additions before each multiply, which means the input operand in the multiplier will be wider than the weight operand. The worst case scenario happens when the activation group size is the entire input tile, i.e., the entire tile corresponds to one unique non-zero weight, in which case the input operand is widest. This case is unlikely in practice, and increases multiplier cost in the common case where the activation group size is $\ll$ input tile size. Therefore, we set a maximum limit for the activation group size. In case the activation group size exceeds the limit, we split activation groups into chunks up to the maximum size. A local counter triggers early MACs along with weight buffer 'peeks' at group boundaries. In this work, we assume a maximum activation group size of 16. This means we can reduce multiplies by $16\times$ in the best case, and the multiplier is 4 bits wider on one input.

C. Activation Group Reuse

We now augment the above architecture to exploit activation group reuse (Section III-B). The key idea is that by carefully ordering the entries in the input indirection table (iiT), a single input indirection table can be shared across multiple filters. This has two benefits. First, we reduce the model size since the total storage needed for the input indirection tables shrinks. Second, with careful engineering, we can share sub-computations between the filters, which saves input buffer reads and improves PE throughput. Recall that the number of filters sharing the same indirection table is a parameter G, as noted in Table I. If $G = 1$, we have vanilla dot product factorization from the previous section.

Indirection table hierarchical sorting. To support $G > 1$, we hierarchically sort a single input indirection table to support G filters. Due to the hierarchical sort, we will still be able to implement the weight indirection tables as a single bit per entry per filter, as done in Section IV-B. We give an example for the $G = 2$ case in Figure 7, and walk through how to hierarchically sort the indirection tables below. These steps are performed offline.

1) Select a canonical order of weights. The order is a, b in the example.
2) Sort entries by activation group for the first filter k_1.
3) Within each activation group of k_1, sort by sub-activation group for the second filter k_2 using the *same* canonical
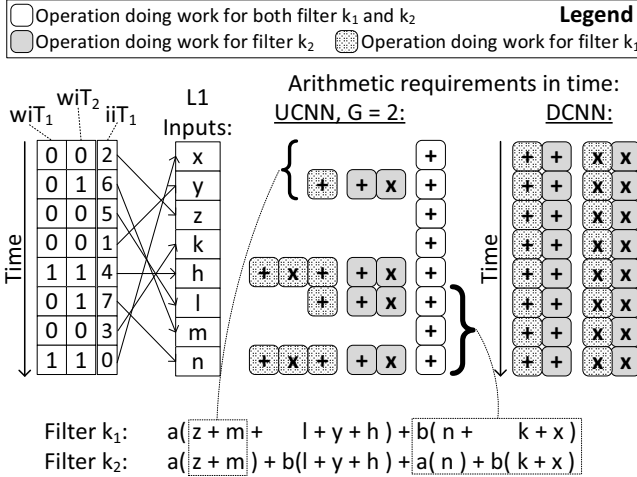
Fig. 7. Example of activation group reuse for $G = 2$ with weights a, b. The indirection tables iiT and wiT are walked top to bottom (time moves down). At each step, the sequence of adds and multiplies needed to evaluate that step are shown right to left. Recall a MAC is a multiply followed by an add. We assume that at the start of building each sub-activation group, the state for accumulator ② is reset to 0. As shown, DCNN with two DCNN lanes processes these filters with 16 multiplies, whereas UCNN completes the same work in 6 multiplies.

order a, b. Filter k_1 has activation groups (e.g., $z + m + l + y + h$) and filter k_2 has sub-activation groups within filter k_1's groups (e.g., $z + m$ and $l + y + h$).

Now, a single traversal of the input indirection table can efficiently produce results for both filters k_1 and k_2. Crucially, sorts performed in Step 2 and 3 are *keyed to the same canonical order of weights* we chose in Step 1 (a, b in the example). By keeping the same order across filters, the weight indirection tables (denoted wiT$_1$ and wiT$_2$ for k_1 and k_2, respectively, in Figure 7) can be implemented as a single bit per entry.

As mentioned above, the scheme generalizes to $G > 2$ in the natural fashion. For example, for $G = 3$ we additionally sort sub-sub-activation groups within the already established sub-activation groups using the same canonical a, b weight order. Thus, the effective indirection table size per weight is $(|\text{iiT.entry}| + G * |\text{wiT.entry}|)/G = \lceil \log_2 RSC_t \rceil / G + 1$ which is an $O(G)$ factor compression. We will see the upper bound for G later in this section.

Datapath. To support activation group reuse, we add a third accumulator to the PE to enable accumulations across different level activation groups (Figure 6 ③). G-th activation groups are first summed in accumulator ②. At G-th level activation group boundaries, the G-th level sum is merged into running sums for levels $g = 1, \ldots, G-1$ using accumulator ③. At any level activation group boundary, sums requiring a multiply are dispatched to the MAC unit ①.

For clarity, we now give a step-by-step (in time) description of this scheme using the example in Figure 7 and the architecture from Figure 6. Recall, we will form activation groups for filter k_1 and sub-activation groups for filter k_2.

1) The input indirection table iiT reads the indirection to be 2, which corresponds to activation z. This is sent to accumulator ② which starts building the sub-activation group containing z. We assume accumulator ②'s state is reset at the start of each sub-activation group, so the accumulator implicitly calculates $0 + z$ here. Both wiT$_1$ and wiT$_2$ read 0s, thus we proceed without further accumulations.

2) iiT reads 6 and wiT$_1$ and wiT$_2$ read 0 and 1, respectively. This means we are at the end of the sub-activation group (for filter k_2), but not the activation group (for filter k_1). Sum $z + m$ is formed in accumulator ②, which is sent (1) to accumulator ③—as this represents the sum of only a part of the activation group for filter k_1—and (2) to the MAC unit ① to multiply with a for filter k_2.

3) Both wiT$_1$ and wiT$_2$ read 0s, accumulator ② starts accumulating the sub-activation group containing l.

4) Both wiT$_1$ and wiT$_2$ read 0s, accumulator ② builds $l + y$.

5) Both wiT$_1$ and wiT$_2$ read 1s, signifying the end of both the sub-activation and activation groups. Accumulator ② calculates $l + y + h$, while accumulator ③ contains $z + m$ for filter k_1. The result from accumulator ② is sent (1) to the MAC Unit ①—to multiply with b for filter k_2—and (2) to accumulator ③ to generate $z + m + l + y + h$. The result from accumulator ③ finally reaches the MAC Unit ① to be multiplied with a.

6) Repeat steps similar to those shown above for subsequent activation groups on filter k_1, until the end of the input indirection table traversal.

Together, we refer to all of the above arithmetic and control as a *UCNN lane*. Note that a transition between activation groups in k_1 implies a transition for k_2 as well.

Area implications. To vectorize by a factor of G, a dense design requires G multipliers. However, as shown in Figure 6, we manage to achieve similar throughput with a single multiplier. The multiplier reduction is possible because the multiplier is only used on (sub-)activation group transitions. We do note that under-provisioning multipliers can lead to stalls, e.g., if (sub-)activation group transitions are very common. Thus, how many hardware multipliers and accumulators to provision is a design parameter. We evaluate a single-multiplier design in Section VI-C.

Handling empty sub-activation groups. In Figure 7, if weight a or b in filters k_1 or k_2 had a (sub-)activation group size of zero, the scheme breaks because each filter cycles through weights in the same canonical order. To properly handle these cases, we have two options. First, we can allocate more bits per entry in the weight indirection table. That is, interpret weight indirection table entries as n-bit counters that can skip 0 to $2^n - 1$ weights per entry. Second, we can add special "skip" entries to the weight and input indirection tables to skip the weight without any computations. A simple skip-entry design would create a cycle bubble in the UCNN lane per skip.

We apply a hybrid of the above schemes in our implementation. We provision an extra bit to each entry in the G-th filter's weight indirection table, for each group of G filters. An extra

680

bit enables us to skip up to 3 weights. We find we only need to add a bit to the G-th filter, as this filter will have the smallest activation groups and hence has the largest chance of seeing an empty group. For any skip distance longer than what can be handled in allocated bits, we add skip entries as necessary and incur pipeline bubbles.

Additional table compression. We can further reduce the bits per entry in the input indirection table by treating each entry as a jump, relative to the last activation sharing the same weight, instead of as a direct pointer. This is similar to run-length encodings (RLEs) in sparse architectures [27], [11], [12]. Represented as jumps, bits per table entry are proportional to the average distance between activations sharing the same weight (i.e., $O(\log_2 U)$), which can be smaller than the original pointer width $\lceil \log_2 RSC_t \rceil$. The trade-off with this scheme is that if the required jump is larger than the bits provisioned, we must add skip entries to close the distance in multiple hops.[1]

Activation group reuse implications for weight sparsity. Fundamentally, to service G filters we need to read activations according to the *union* of non-zero weights in the group of G filters. That is, we can only remove entries from indirection tables if the corresponding weight in filters k_1 *and* k_2 is 0. Thus, while we get an $O(G)$ factor of compression in indirection tables, less entries will be skip-able due to weight sparsity.

D. Spatial Vectorization

One overhead unique to the UCNN PE is the cost to indirect into the input buffer. The indirection requires an extra buffer access, and the irregular access pattern means the input SRAM cannot read out vectors (which increases pJ/bit). Based on the observation that indirection tables are reused for every filter slide, we propose a novel method to vectorize the UCNN PE across the spatial WH dimensions. Such reuse allows UCNN to amortize the indirection table lookups across vector lanes. We refer to this scheme as *spatial vectorization* and introduce a new parameter V_W to indicate the spatial vector size.

To implement spatial vectorization, we split the input buffer into V_W banks and carefully architect the buffer so that exactly V_W activations can be read every cycle. We note the total input buffer capacity required is only $O(C_t * S * (V_W + R))$, not $O(C_t * S * V_W * R)$, owing to the overlap of successive filter slides. The datapath for activation group reuse (Section IV-C) is replicated across vector lanes, thus improving the PE throughput to $O(G * V_W)$ relative to the baseline non-vectorized PE. Given that UCNN significantly reduces multiplier utilization, an aggressive implementation could choose to temporally multiplex $< V_W$ multipliers instead of spatially replicating multipliers across lanes.

Avoiding bank conflicts. Since the input buffer access pattern is irregular in UCNN, there may be bank conflicts in the banked input buffer. To avoid bank conflicts, we divide the input buffer into V_W banks and apply the following fill/access strategy. To evaluate V_W dot products, we iterate through the input buffer according to the input indirection table. We denote each

indirection as a tuple $(r, s, c) \in RSC_t$, where (r, s, c) corresponds to the spatial vector base address. Then, the bank id/bank address to populate vector slot $v \in [0, \ldots, V_W - 1]$ for that indirection is:

$$\text{bank}(r, s, c, v) = (r + v) \% V_W \tag{3}$$

$$\text{addr}(r, s, c, v) = s * C_t + c + \lceil (r + v)/V_W \rceil * S * C_t \tag{4}$$

This strategy is bank conflict free because $\text{bank}(r, s, c, v)$ always yields a different output for fixed (r, s, c), varying v. Unfortunately, this scheme has a small storage overhead: a $((R + V_W - 1) \% V_W)/(R + V_W - 1)$ fraction of addresses in the input buffer are un-addressable. Note, this space overhead is always $< 2\times$ and specific settings of R and V_W can completely eliminate overhead (e.g., $V_W = 2$ for $R = 3$).

E. UCNN Design Flexibility

Supporting a range of U. Based on the training procedure, CNNs may have a different number of unique weights (e.g., 3 [17] or 17 [18] or 256 [14] or more). Our accelerator can flexibly handle a large range of U, but still gain the efficiency in Section IV-A, by reserving a larger weight buffer in the PE. This enables UCNN to be used on networks that are not re-trained for quantization as well. We note that even if U is large, we still save energy by removing redundant weight buffer accesses.

Support for other layer types. CNNs are made up of multiple layer types including convolutional, non-linear activation, pooling and fully connected. We perform non-linear activations (e.g., ReLu [28]) at the PE (see Figure 8 (F)). Pooling can be handled with minimal additional logic (e.g., max circuits) at the PE, with arithmetic disabled. We implement fully connected layers as convolutions where input buffer slide reuse is disabled (see next section).

V. ARCHITECTURE AND DATAFLOW

This section presents the overall architecture for DCNN and UCNN, i.e., components beyond the PEs, as well as the architecture's dataflow. CNN dataflow [27] describes how and when data moves through the chip. We present a dataflow that both suits the requirements of UCNN and provides the best power efficiency and performance out of candidates that we tried.

As described in the previous section and in Figure 5, the DCNN and UCNN architectures consist of multiple Processing Elements (PEs) connected to a shared global buffer (L2), similar to previous proposals [9], [27], [19]. Similar to the PEs, the L2 buffer is divided into input and weight buffers. When it is not clear from context, we will refer to the PE-level input and weight buffers (Section IV) as the L1 input and weight buffers. Each PE is fed by two multicast buses, for input and weight data. Final output activations, generated by PEs, are written back to the L2 alongside the input activations in a double-buffered fashion. That is, each output and will be treated as an input to the next layer.

A. Dataflow

Our dataflow is summarized as follows. We adopt weight- and output-stationary terminology from [27].

1) The design is weight-stationary at the L2, and stores all input activations on chip when possible.
2) Each PE produces one column of output activations and PEs work on adjacent overlapped regions of input. The overlapping columns create input halos [12].
3) Each PE is output-stationary, i.e., the partial sum resides in the PE until the final output is generated across all *C* input channels.

At the top level, our dataflow strives to minimize reads/writes to DRAM as DRAM often is the energy bottleneck in CNN accelerators [19], [12]. Whenever possible, we store all input activations in the L2. We do not write/read input activations from DRAM unless their size is prohibitive. We note that inputs fit on chip in most cases, given several hundred KB of L2 storage.[2] In cases where inputs fit, we only need to read inputs from DRAM once, during the first layer of inference. In cases where inputs do not fit, we tile the input spatially. In all cases, we read all weights from DRAM for every layer. This is fundamental given the large (sometimes 10s of MB) aggregate model size counting all layers. To minimize DRAM energy from weights, the dataflow ensures that each weight value is fetched a minimal number of times, e.g., once if inputs fit and once per input tile otherwise.

At the PE, our dataflow was influenced by the requirements of UCNN. Dot product factorization (Section III) builds activation groups through *RSC* regions, hence the dataflow is designed to give PEs visibility to *RSC* regions of weights and inputs in the inner-most (PE-level) loops. We remark that dataflows working over *RSC* regions in the PEs have other benefits, such as reduced partial sum movement [27], [9].

Detailed pseudo-code for the complete dataflow is given in Figure 8. For simplicity, we assume the PE is not vectorized. Inputs reside on-chip, but weights are progressively fetched from DRAM in chunks of K_c filters at a time (A). K_c may change from layer to layer and is chosen such that the L2 is filled. Work is assigned to the PEs across columns of input and filters within the K_c-size group (B). Columns of input and filters are streamed to PE-local L1 buffers (C). Both inputs and weights may be multicast to PEs (as shown by `#multicast`), depending on DNN layer parameters. As discussed in Section IV-A, C_t input channels-worth of inputs and weights are loaded into the PE at a time. As soon as the required inputs/weights are available, RSC_t sub-regions of input are transferred to smaller L0 buffers for spatial/slide data reuse and the dot product is calculated for the RSC_t-size tile (E). Note that each PE works on a column of input of size *RHC* and produces a column of output of size *H* (D). The partial sum produced is stored in the L1 partial sum buffer and the final output is written back to the L2 (F). Note that the partial sum resides in the PE until the final output is generated, making the PE dataflow output-stationary.

[2]For example, all but several ResNet-50 [16] layers can fit inputs on chip with 256 KB of storage and 8 bit activations.

```
def CNNLayer():
  BUFFER in_L2 [C][H][W];
  BUFFER out_L2[K][H][W];
  BUFFER wt_L2 [Kc][C][S][R];
(A)  for kc = 0 to K/Kc - 1
    {
      wt_L2 = DRAM[kc*Kc:(kc+1)Kc-1]
                  [:][:][:];
      #parallel
(B)    for col, k in (col = 0 to W-R) x
                     (k = 0 to Kc-1)
      {
        PE(col, k);
      }
    }

def PE(col, k):
  // col: which spatial column
  // k: filter
  BUFFER in_L1  [Ct][S][R];
  BUFFER psum_L1[H];
  BUFFER wt_L1  [Ct][S][R];
  psum_L1.zero(); // reset psums
(C)  for ct = 0 to C/Ct - 1
    {
      #multicast
      wt_L1 = wt_L2[k][ct*Ct:(ct+1)Ct-1]
                  [:][:];
(D)    for h = 0 to H - S
      {
        // slide reuse for in_L1 not shown
        #multicast
        in_L1 = in_L2[ct*Ct:(ct+1)Ct-1]
                    [h:h+S-1]
                    [col:col+R-1];
        sum = psum_L1[h];
(E)      for r,c,s in (r = 0 to R-1) x
                     (c = 0 to Ct-1) x
                     (s = 0 to S-1)
        {
          act = in_L1[c][s][r];
          wt = wt_L1[c][s][r];
          sum += act * wt;
        }
        psum_L1[h] = sum;
      }
    }
(F)  out_L2[k][:][col] = RELU(psum_L1);
```

Fig. 8. DCNN/UCNN dataflow, parameterized for DCNN (Section IV-A). For simplicity, the PE is not vectorized and stride is assumed to be 1. `[x:y]` indicates a range; `[:]` implies all data in that dimension.

VI. EVALUATION

A. Methodology

Measurement setup. We evaluate UCNN using a whole-chip performance and energy model, and design/synthesize the DCNN/UCNN PEs in RTL written in Verilog. All designs are evaluated in a 32 nm process, assuming a 1 GHz clock. For the energy model, energy numbers for arithmetic units are taken from [29], scaled to 32 nm. SRAM energies are taken from CACTI [30]. For all SRAMs, we assume `itrs-lop` as this

decreases energy per access, but still yields SRAMs that meet timing at 1 GHz. DRAM energy is counted at 20 pJ/bit [29]. Network on chip (NoC) energy is extrapolated based on the number and estimated length of wires in the design (using our PE area and L2 SRAM area estimates from CACTI). We assume the NoC uses low-swing wires [31], which are low power, however consume energy each cycle (regardless of whether data is transferred) via differential signaling.

Activation/weight data types. Current literature employs a variety of activation/weight precision settings. For example, 8 to 16 bit fixed point [12], [11], [9], [13], [14], 32 bit floating point/4 bit fixed point activations with power of two weights [18] to an un-specified (presumably 16 bit fixed point) precision [17]. Exploiting weight repetition is orthogonal to which precision/data type is used for weights and activations. However, for completeness, we evaluate both 8 bit and 16 bit fixed point configurations.

Points of comparison. We evaluate the following design variants:

DCNN: Baseline DCNN (Section IV-A) that does not exploit weight or activation sparsity, or weight repetition. We assume that DCNN is vectorized across output channels and denote the vector width as V_k. Such a design amortizes the L1 input buffer cost and improves DCNN's throughput by a factor of V_k.

DCNN_sp: DCNN with Eyeriss-style [27] sparsity optimizations. DCNN_sp skips multiplies at the PEs when an operand (weight or activation) is zero, and compresses data stored in DRAM with a 5 bit run-length encoding.

UCNN_Uxx: UCNN, with all optimizations enabled (Section IV-C) except for the jump-style indirection table (Section IV-C) which we evaluate separately in Section VI-D. UCNN reduces DRAM accesses based on weight sparsity and activation group reuse, and reduces input memory reads, weight memory reads, multiplies and adds per dot product at the PEs. UCNN also vectorizes spatially (Section IV-D). The _Uxx refers to the number of unique weights; e.g., UCNN_U17 is UCNN with $U = 17$ unique weights, which corresponds to an INQ-like quantization.

CNNs evaluated. To prove the effectiveness of UCNN across a range of contemporary CNNs, we evaluate the above schemes on three popular CNNs: a LeNet-like CNN [20] trained on CIFAR-10 [20], and AlexNet [6] plus ResNet-50 [16] trained on ImageNet [22]. We refer to these three networks as LeNet, AlexNet and ResNet for short.

B. Energy Analysis

We now perform a detailed energy analysis and design space exploration comparing DCNN and UCNN.

Design space. We present results for several weight density points (the fraction of weights that are non-zero), specifically 90%, 65% and 50%. For each density, we set (100-density)% of weights to 0 and set the remaining weights to non-zero values via a uniform distribution. Evaluation on a real weight distribution from INQ training is given in Section VI-C. 90%

Design	P	V_K	V_W	G	L1 inp.	L1 wt.
DCNN	32	8	1	1	144	1152
DCNN_sp	32	8	1	1	144	1152
UCNN ($U = 3$)	32	1	2	4	768	129
UCNN ($U = 17$)	32	1	4	2	1152	232
UCNN ($U > 17$)	32	1	8	1	1920	652

density closely approximates our INQ data. 65% and 50% density approximates prior work, which reports negligible accuracy loss for this degree of sparsification [14], [17], [12]. We note that UCNN does not alter weight values, hence UCNN run on prior training schemes [14], [18], [17] results in the same accuracy as reported in those works. Input activation density is 35% (a rough average from [12]) for all experiments. We note that lower input activation density favors DCNN_sp due to its multiplication skipping logic.

To illustrate a range of deployment scenarios, we evaluate UCNN for different values of unique weights: $U = 3, 17, 64, 256$. We evaluate UCNN_U3 ("TTQ-like" [17]) and UCNN_U17 ("INQ-like" [18]) as these represent two example state-of-the-art quantization techniques. We show larger U configurations to simulate a range of other quantization options. For example, UCNN_U256 can be used on out-of-the-box (not re-trained) networks quantized for 8 bit weights [13] or on networks output by Deep Compression with 16 bit weights [14].

Hardware parameters. Table II lists all the hardware parameters used by different schemes in this evaluation. To get an apples-to-apples performance comparison, we equalize "effective throughput" across the designs in two steps. First, we give each design the same number of PEs. Second, we vectorize each design to perform the work of 8 dense multiplies per cycle per PE. Specifically, DCNN uses $V_K = 8$ and UCNN uses V_W and G such that $G * V_W = 8$, where V_W and V_K represent vectorization in the spatial and output channel dimensions, respectively. Note that to achieve this throughput, the UCNN PE may only require V_W or fewer multipliers (Section IV-C). Subject to these constraints, we allow each design variant to choose a different L1 input buffer size, V_W and G to maximize its own average energy efficiency.

Results. Figure 9 shows energy consumption for three contemporary CNNs at both 8 and 16 bit precision. Energy is broken into DRAM, L2/NoC and PE components. Each configuration (for a particular network, weight precision and weight density) is normalized to DCNN for that configuration.

At 16 bit precision, all UCNN variants reduce energy compared to DCNN_sp. The improvement comes from three sources. First, activation group reuse ($G > 1$ designs in Table II) reduces DRAM energy by sharing input indirection tables across filters. Second, activation group reuse (for any G) reduces energy from arithmetic logic at the PE. Third, decreasing
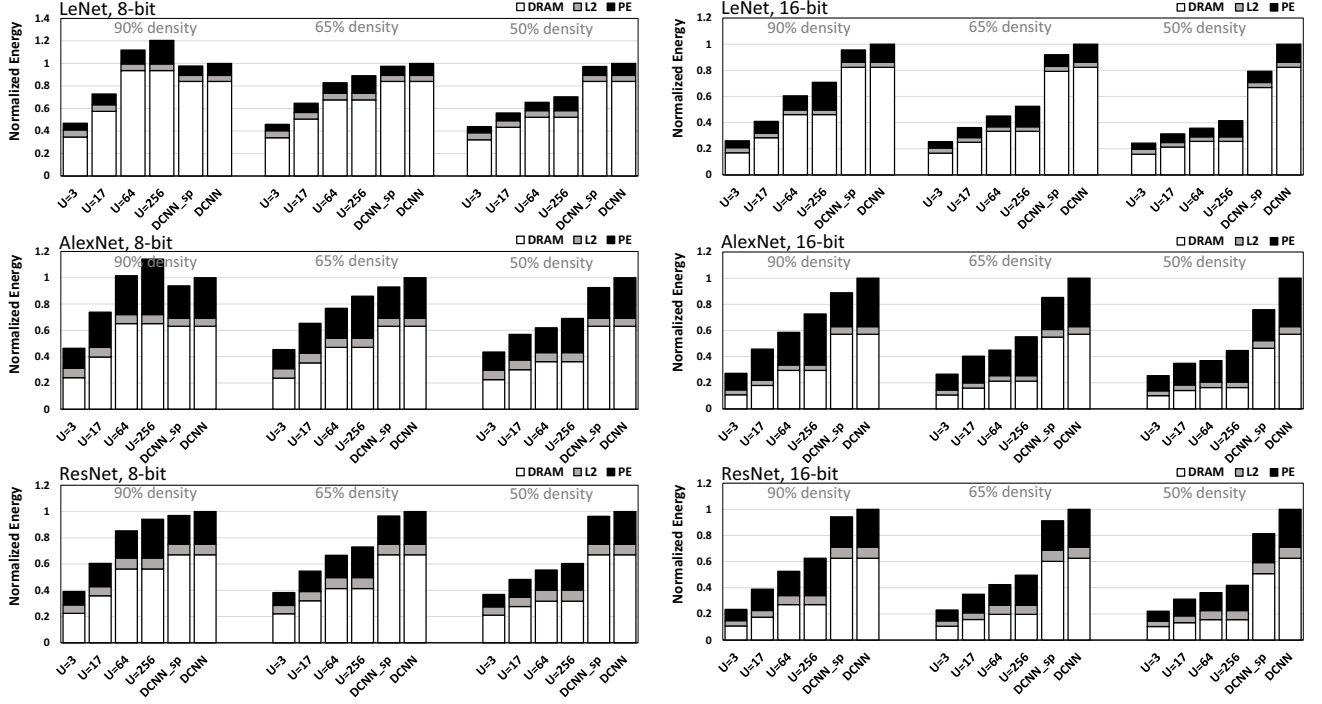
683

Fig. 9. Energy consumption analysis of the three popular CNNs discussed in Section VI-A, running on UCNN and DCNN variants. UCNN variant UCNN_Uxx is shown as $U = xx$. Left and right graphs show results using 8 bit and 16 bit weights, respectively. For each configuration, we look at 90%, 65% and 50% densities. In all cases, input density is 35%. Each group of results (for a given network and weight precision/density) is normalized to the DCNN configuration in that group.

weight density results in fewer entries per indirection table on average, which saves DRAM accesses and cycles to evaluate each filter. Combining these effects, UCNN_U3, UCNN_U17 and UCNN_U256 reduce energy by up to $3.7\times$, $2.6\times$ and $1.9\times$, respectively, relative to DCNN_sp for ResNet-50 at 50% weight density. We note that 50% weight density improves DCNN_sp's efficiency since it can also exploit sparsity. Since DCNN cannot exploit sparsity, UCNN's improvement widens to $4.5\times$, $3.2\times$ and $2.4\times$ compared to DCNN, for the same configurations. Interestingly, when given relatively dense weights (i.e., 90% density as with INQ training), the UCNN configurations attain a $4\times$, $2.4\times$ and $1.5\times$ improvement over DCNN_sp. The improvement for UCNN_U3 increases relative to the 50% dense case because DCNN_sp is less effective in the dense-weights regime.

We observed similar improvements for the other networks (AlexNet and LeNet) given 16 bit precision, and improvement across all networks ranges between $1.2\times \sim 4\times$ and $1.7\times \sim 3.7\times$ for 90% and 50% weight densities, respectively.

At 8 bit precision, multiplies are relatively cheap and DRAM compression is less effective due to the relative size of compression metadata. Thus, improvement for UCNN_U3, UCNN_U17 and UCNN_U256 drops to $2.6\times$, $2\times$ and $1.6\times$, respectively, relative to DCNN_sp on ResNet-50 and 50% weight density. At the 90% weight density point, UCNN variants with $U = 64$ and $U = 256$ perform worse than DCNN_sp on AlexNet and LeNet. These schemes use $G = 1$
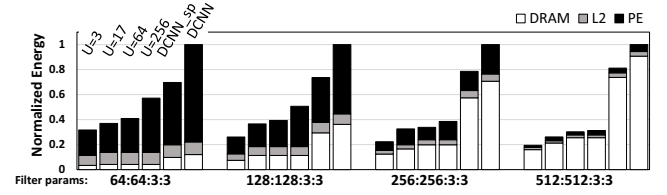

Fig. 10. Energy breakdown for the 50% weight density and 16 bit precision point, for specific layers in ResNet. Each group of results is for one layer, using the notation $C : K : R : S$. All results are relative to DCNN for that layer.

and thus incur large energy overheads from reading indirection tables from memory. We evaluate additional compression techniques to improve these configurations in Section VI-D.

To give additional insight, we further break down energy by network layer. Figure 10 shows select layers in ResNet-50 given 50% weight density and 16 bit precision. Generally, early layers for the three networks (only ResNet shown) have smaller C and K; later layers have larger C and K. DRAM access count is proportional to total filter size $R * S * C * K$, making early and later layers compute and memory bound, respectively. Thus, UCNN reduces energy in early layers by improving arithmetic efficiency and reduces energy in later layers by saving DRAM accesses.

C. Performance Analysis

We now compare the performance of UCNN to DCNN with the help of two studies. First, we compare performance assuming no load balance issues (e.g., skip entries in indirection tables; Section IV-C) and assuming a uniform distribution of weights across filters, to demonstrate the benefit of sparse weights. Second, we compare performance given real INQ [18] data, taking into account all data-dependent effects. This helps us visualize how a real implementation of UCNN can differ from the ideal implementation. For all experiments, we assume the hardware parameters in Table II.
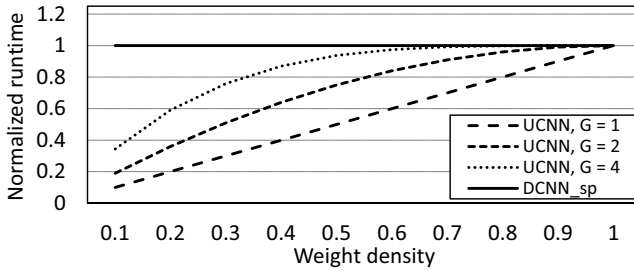

Fig. 11. Normalized runtime in cycles (lower is better) between DCNN_sp and UCNN variants. Runtimes are normalized to DCNN_sp.

Optimistic performance analysis. While all designs in Table II are throughput-normalized, UCNN can still save cycles due to weight sparsity as shown in Figure 11. Potential improvement is a function of G: as described in Section IV-C, the indirection tables with activation group reuse ($G > 1$) must store entries corresponding to the union of non-zero weights across the G filters. This means that choosing G presents a performance energy trade-off: larger G (when this is possible) reduces energy per CNN inference, yet smaller G (e.g., $G = 1$) can improve runtime.

Performance on real INQ data. We now compare UCNN to DCNN on real INQ [18] training data ($U = 17$) and take into account sources of implementation-dependent UCNN performance overhead (e.g., a single multiplier in the PE datapath, and table skip entries; Section IV-C). The result is presented in Figure 12. Given that our model trained with INQ has 90% weight density (matching [18]), UCNN could improve performance by 10% in the best case (Section VI-B). However, we see 0.7% improvement for UCNN ($G = 1$). We further observe the following: increasing $V_K = 2$ for DCNN_sp, DCNN's performance improves by $2\times$. However, UCNN $G = 2$ (which is throughput-normalized to DCNN $V_K = 2$) only improves performance by $1.80\times$, deviating from the ideal improvement of $2\times$. This performance gap is largely due to skip entries in the indirection table (Section IV-C). Overall, the performance deficit is dominated by the energy savings with UCNN as presented in Section VI-B. Therefore, UCNN still provides a significant performance/watt advantage over DCNN configurations.

D. Model Size (DRAM storage footprint)

Figure 13 compares weight compression rates between UCNN variants, DCNN_sp and to the stated model sizes in the TTQ [17] and INQ [18] papers. UCNN uses activation group reuse and weight sparsity to compress model size (Section IV-C), however uses the simple pointer scheme from Section IV-B to minimize skip entries. DCNN_sp uses a run-length encoding as discussed in Section VI-A. TTQ [17] and INQ [18] represent weights as 2-bit and 5-bit indirections, respectively. UCNN, TTQ and INQ model sizes are invariant to the bit-precision per weight. This is not true for DCNN_sp, so we only show DCNN_sp with 8 bits per weight to make it more competitive. TTQ and INQ cannot reduce model size further due to weight sparsity: e.g., a run-length encoding would outweigh the benefit because their representation is smaller than the run-length code metadata.

UCNN models with $G > 1$ are significantly smaller than DCNN_sp for all weight densities. However, UCNN $G = 1$ (no activation group reuse) results in a larger model size than DCNN_sp for models with higher weight density.

We now compare UCNN's model size with that of TTQ and INQ. At the 50% weight density point, UCNN $G = 4$ (used for TTQ) requires ~ 3.3 bits per weight. If density drops to 30%, model size drops to < 3 bits per weight, which [17] shows results in $\sim 1\%$ accuracy loss. At the 90% weight density point, UCNN $G = 2$ (used for INQ) requires 5-6 bits per weight. Overall, we see that UCNN model sizes are competitive with the best known quantization schemes, and simultaneously give the ability to reduce energy on-chip.

Effect of jump-based indirection tables. Section IV-C discussed how to reduce model size for UCNN further by replacing the pointers in the input indirection table with jumps. The downside of this scheme is possible performance overhead: if the jump width isn't large enough, multiple jumps will be needed to reach the next weight which results in bubbles. We show these effects on INQ-trained ResNet in Figure 14. There are two takeaways. First, in the $G = 1$ case, we can shrink the bits/weight by 3 bits (from 11 to 8) without incurring serious performance overhead ($\sim 2\%$). In that case, the $G = 1$ point never exceeds the model size for DCNN_sp with 8 bit weights. Second, for the $G = 2$ case we can shrink the bits/weight by 1 bit (from 6 to 5), matching INQ's model size with negligible performance penalty. We note that the same effect can be achieved if the INQ model weight density drops below 60%.

E. Hardware RTL Results

Finally, Table VI-E shows the area overhead of UCNN mechanisms at the PE. We implement both DCNN and UCNN PEs in Verilog, using 16 bit precision weights/activations. Synthesis uses a 32 nm process, and both designs meet timing at 1 GHz. Area numbers for SRAM were obtained from CACTI [30] and the area for logic comes from synthesis. For a throughput-normalized comparison, and to match the performance study in Section VI-C, we report area numbers for the DCNN PE with $V_K = 2$ and the UCNN PE with $G = 2, U = 17$.
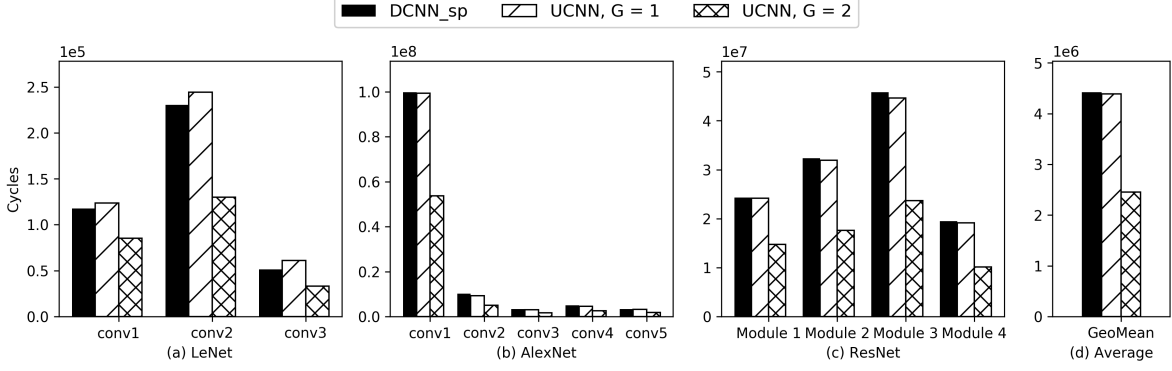
685

Fig. 12. Performance study, comparing DCNN_sp ($V_K = 1$) and UCNN variants on the three networks from Section VI-A. The geometric means for all variants are shown in (d).
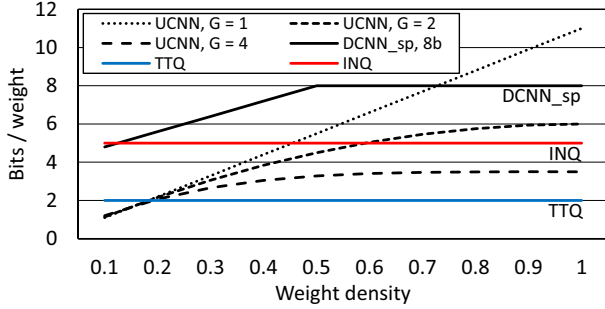

Fig. 13. Model size (normalized per weight), as a function of weight density. UCNN indirection table entries are pointers.
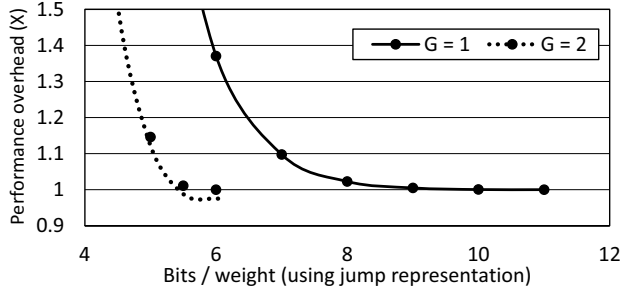

Fig. 14. UCNN model size (normalized per weight), decreasing jump entry width, for the INQ-trained ResNet.

TABLE III
UCNN PE AREA BREAKDOWN (IN mm^2).

Component	DCNN($V_K = 2$)	UCNN ($G = 2, U = 17$)
Input buffer	0.00135	0.00453
Indirection table	—	0.00100
Weight buffer	0.00384	—
Partial Sum buffer	0.00577	0.00577
Arithmetic	0.00120	0.00244
Control Logic	0.00109	0.00171
Total	0.01325	0.01545

Provisioned with a weight buffer F of 17 entries, the UCNN PE adds 17% area overhead compared to a DCNN PE. If we provision for 256 weights to improve design flexibility (Section IV-E), this overhead increases to 24%. Our UCNN design multiplexes a single MAC unit between $G = 2$ filters and gates the PE datapath when the indirection table outputs a skip entry (Section VI-C). The RTL evaluation reproduces the performance results from our performance model (Section VI-C).

VII. RELATED WORK

Weight quantization. There is a rich line of work that studies DNN machine efficiency-result accuracy trade-offs by skipping zeros in DNNs and reducing DNN numerical precision (e.g., [14], [18], [32], [17]). Deep Compression [14], INQ [18] and TTQ [17] achieve competitive accuracy on different networks trained on Imagenet [22], although we note that TTQ loses several percent accuracy on ResNet [16]. Our work strives to support (and improve efficiency for) all of these schemes in a precision and weight-quantization agnostic fashion.

Sparsity and sparse accelerators. DNN sparsity was first recognized by Optimal Brain Damage [33] and more recently was adopted for modern networks in Han et al. [25], [14]. Since then, DNN accelerators have sought to save cycles and energy by exploiting sparse weights [19], activations [10] or both [12], [11]. Relative to our work, these works exploit savings though repeated *zero* weights, whereas we exploit repetition in zero *or non-zero* weights. As mentioned, we gain additional efficiency through weight sparsity.

Algorithms to exploit computation re-use in convolutions. Reducing computation via repeated weights draws inspiration from the Winograd style of convolution [34]. Winograd factors out multiplies in convolution (similar to how we factorized dot products) by taking advantage of the predictable filter slide. Unlike weight repetition, Winograd is weight/input "repetition un-aware", can't exploit cross-filter weight repetition, loses effectiveness for non-unit strides and only works for convolutions. Depending on quantization, weight repetition architectures can exploit more opportunity. On the other hand,

Winograd maintains a more regular computation and is thus more suitable for general purpose devices such as GPUs. Thus, we consider it important future work to study how to combine these techniques to get the best of both worlds.

TTQ [17] mentions that multiplies can be replaced with a table lookup (code book) indexed by activation. This is similar to partial produce reuse (Section III-C), however faces challenges in achieving net efficiency improvements. For example: an 8 bit and 16 bit fixed point multiply in 32 nm is .1 and .4 pJ, respectively. The corresponding table lookups (512-entry 8 bit and 32K-entry 16 bit SRAMs) cost .17 and 2.5 pJ, respectively [30]. Thus, replacing the multiplication with a lookup actually increases energy consumption. Our proposal gets a net-improvement by reusing compound expressions.

Architectures that exploit repeated weights. Deep compression [14] and EIE [11] propose weight sharing (same phenomena as repeated weights) to reduce weight storage, however do not explore ways to reduce/re-use sub computations (Section III) through shared weights. Further, their compression is less aggressive, and doesn't take advantage of overlapped repetitions across filters.

VIII. Conclusion

This paper proposed UCNN, a novel CNN accelerator that exploits weight repetition to reduce on-chip multiplies/memory reads and to compress network model size. Compared to an Eyeriss-style CNN accelerator baseline, UCNN improves energy efficiency up to $3.7\times$ on three contemporary CNNs. Our advantage grows to $4\times$ when given dense weights. Indeed, we view our work as a first step towards generalizing sparse architectures: we should be exploiting repetition in all weights, not just zero weights.

IX. Acknowledgements

References

[1] G. Hinton, L. Deng, D. Yu, G. Dahl, A. rahman Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, B. Kingsbury, and T. Sainath, "Deep neural networks for acoustic modeling in speech recognition," *IEEE Signal Processing Magazine*, vol. 29, pp. 82–97, November 2012.

[2] D. Ciregan, U. Meier, and J. Schmidhuber, "Multi-column deep neural networks for image classification," CVPR'12.

[3] J. Morajda, "Neural networks and their economic applications," in *Artificial intelligence and security in computing systems*, pp. 53–62, Springer, 2003.

[4] J. L. Patel and R. K. Goyal, "Applications of artificial neural networks in medical science," *Current clinical pharmacology*, vol. 2, no. 3, pp. 217–226, 2007.

[5] H. Malmgren, M. Borga, and L. Niklasson, *Artificial Neural Networks in Medicine and Biology: Proceedings of the ANNIMAB-1 Conference, Göteborg, Sweden, 13–16 May 2000.* Springer Science & Business Media, 2012.

[6] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems*, NIPS'12.

[7] K. Simonyan and A. Zisserman, "Two-stream convolutional networks for action recognition in videos," NIPS'14.

[8] A. Radford, L. Metz, and S. Chintala, "Unsupervised representation learning with deep convolutional generative adversarial networks," *arXiv preprint arXiv:1511.06434*, 2015.

[9] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, and O. Temam, "Dadiannao: A machine-learning supercomputer," MICRO'14.

[10] J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger, and A. Moshovos, "Cnvlutin: Ineffectual-neuron-free deep neural network computing," ISCA'16.

[11] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "EIE: efficient inference engine on compressed deep neural network," ISCA'16.

[12] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S. W. Keckler, and W. J. Dally, "Scnn: An accelerator for compressed-sparse convolutional neural networks," ISCA'17.

[13] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, R. C. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon, "In-datacenter performance analysis of a tensor processing unit," ISCA'17.

[14] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding," ICLR'16.

[15] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," CVPR'15.

[16] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," CVPR'16.

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

[18] A. Zhou, A. Yao, Y. Guo, L. Xu, and Y. Chen, "Incremental network quantization: Towards lossless cnns with low-precision weights," ICLR'17.

[19] S. Zhang, Z. Du, L. Zhang, H. Lan, S. Liu, L. Li, Q. Guo, T. Chen, and Y. Chen, "Cambricon-x: An accelerator for sparse neural networks," MICRO'16.

[20] "Caffe cifar-10 cnn." https://github.com/BVLC/caffe/blob/master/examples/cifar10/cifar10_quick_train_test.prototxt.

[21] A. Krizhevsky, V. Nair, and G. Hinton, "The cifar-10 dataset," *online: http://www.cs.toronto.edu/kriz/cifar.html*, 2014.

[22] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "ImageNet: A Large-Scale Hierarchical Image Database," CVPR'09.

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

[24] J. Cong and B. Xiao, "Minimizing computation in convolutional neural networks," in *ICANN*, 2014.

[25] S. Han, J. Pool, J. Tran, and W. J. Dally, "Learning both weights and connections for efficient neural networks," NIPS'15.

[26] V. Vanhoucke, A. Senior, and M. Z. Mao, "Improving the speed of neural networks on cpus," in *Proc. Deep Learning and Unsupervised Feature Learning NIPS Workshop*, vol. 1, p. 4, Citeseer, 2011.

[27] Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks," ISCA'16.

[28] A. L. Maas, A. Y. Hannun, and A. Y. Ng, "Rectifier nonlinearities improve neural network acoustic models," ICML'13.

[29] M. Horowitz, "Computing's energy problem (and what we can do about it)." ISSCC, 2014.

[30] N. Muralimanohar and R. Balasubramonian, "Cacti 6.0: A tool to understand large caches," 2009.

[31] A. N. Udipi, N. Muralimanohar, and R. Balasubramonian HiPC'09.

[32] M. Courbariaux and Y. Bengio, "Binarynet: Training deep neural networks with weights and activations constrained to +1 or -1," NIPS'16.

[33] Y. LeCun, J. S. Denker, and S. A. Solla, "Optimal brain damage," in *Advances in Neural Information Processing Systems 2* (D. S. Touretzky, ed.), pp. 598–605, Morgan-Kaufmann, 1990.

[34] A. Lavin and S. Gray, "Fast algorithms for convolutional neural networks," CVPR'16.