JPEG-ACT: Accelerating Deep Learning via Transform-based Lossy Compression

R. David Evans
Electrical and Computer Engineering
University of British Columbia
Vancouver, Canada
rdevans@ece.ubc.ca

Lufei Liu
Electrical and Computer Engineering
University of British Columbia
Vancouver, Canada
lucy.lufei@alumni.ubc.ca

Tor M. Aamodt
Electrical and Computer Engineering
University of British Columbia
Vancouver, Canada
aamodt@ece.ubc.ca

***Abstract*—A reduction in the time it takes to train machine learning models can be translated into improvements in accuracy. An important factor that increases training time in deep neural networks (DNNs) is the need to store large amounts of temporary data during the back-propagation algorithm. To enable training very large models this temporary data can be offloaded from limited size GPU memory to CPU memory but this data movement incurs large performance overheads.**

We observe that in one important class of DNNs, convolutional neural networks (CNNs), there is spatial correlation in these temporary values. We propose JPEG for ACTivations (JPEG-ACT), a lossy activation offload accelerator for training CNNs that works by discarding redundant spatial information. JPEG-ACT adapts the well-known JPEG algorithm from 2D image compression to activation compression. We show how to optimize the JPEG algorithm so as to ensure convergence and maintain accuracy during training. JPEG-ACT achieves $2.4\times$ higher training performance compared to prior offload accelerators, and $1.6\times$ compared to prior activation compression methods. An efficient hardware implementation allows JPEG-ACT to consume less than 1% of the power and area of a modern GPU.

***Index Terms*—GPU, Hardware Acceleration, CNN Training, Compression**

I. Introduction

Reductions in training time of deep neural networks [1] played an important role in enabling dramatic improvements in accuracy [2]. Those accuracy improvements, in turn, led to an explosion in the application of deep learning in recent years. These speedups were due to the use of graphics processor units (GPUs) in place of out-of-order superscalar processor architectures. While many recent papers propose advances in specialized hardware acceleration of networks during inference (after a network has been trained) far less have discussed hardware acceleration of the training process.

In this paper, we focus on accelerating the training of Convolutional Neural Networks (CNNs). CNNs have produced state-of-the-art results in image classification, object detection, and semantic labelling [2]–[5]. Typically when training a CNN the output of each individual neuron, called its activation, is computed, saved to memory and, later restored. Activation values are needed again when updating weights using backpropagation [6]. Saving these activation values requires large memory capacities. For example, ResNet50 [3] trained on the ImageNet dataset [7] requires over 40GB of storage,

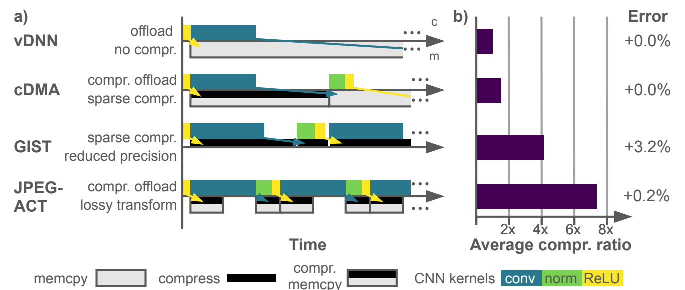

Fig. 1: a) Forward pass offload schedules for repeating conv/norm/ReLU (CNR) blocks in ResNet50/ImageNet. c: compute streams, m: memcpy stream, arrows: corresponding activation offloads. b) Compression ratios on ResNet50/ImageNet, error indicates change from no compression on the validation dataset.

which is greater than the memory available on consumer-grade GPUs (e.g. 12GB, NVIDIA Titan V). State-of-the-art networks contain more layers and larger input image dimensions [3], [8]–[10]. E.g., GPIPE increases memory storage by $4.6\times$ to achieve 10% higher accuracy versus ResNet50 [9].

Cost-effective activation storage can be achieved via recomputation, GPU memory compression, and transfer to CPU-attached memory. Recomputing activations in the backward pass incurs compute overhead [11], [12]. Memory compression has been evaluated on GPUs and activations (GIST, Figure 1) [13] and is well studied on CPUs [14]–[18] but is still limited by the amount of GPU memory. Naively offloading activations to CPU DRAM (e.g., vDNN in Figure 1) [19] or disaggregated memory [20] is limited by PCIe throughput or requires expensive specialized interconnects (e.g., NVLINK on the IBM Power9). However, the cost effectiveness of offloading can be enhanced by compressing the data before it is transferred [21] (cDMA, Figure 1). We build upon the latter approach as it allows for lower cost interconnect and memory technologies (JPEG-ACT, Figure 1).

Activation compression for CPU offload has been studied for shallow networks containing high sparsity [13], [19], [21]. However, during training, ResNets and other extremely deep networks have a high proportion of dense activations and

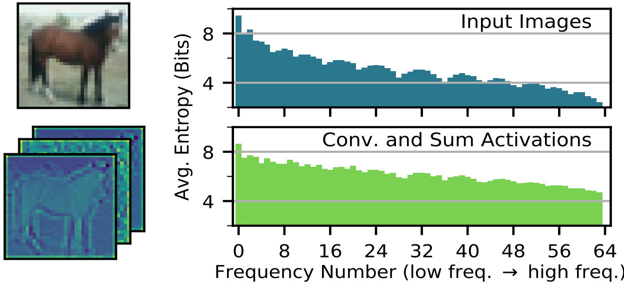

Fig. 2: Frequency entropy distribution for images and non-sparse ResNet50/CIFAR10 activations. Measured using Shannon Entropy of a Discrete Cosine Transform.

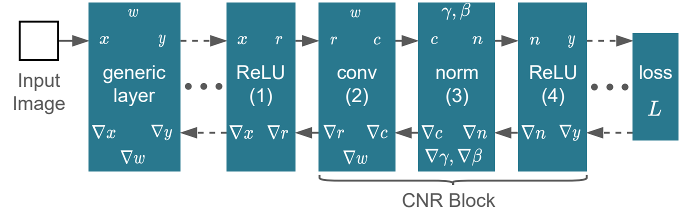

Fig. 3: Training a Convolutional Neural Network using backprop. conv: convolution, norm: batch normalization, ReLU: Rectified Linear Unit

sparse activations (average sparsity of $\approx$ 50%) causing large performance penalties. The sparse methods used by GIST [13] and vDNN [21] perform best when activation sparsity is high, and have a maximum dense compression ratio of 4×.

We propose JPEG-ACT, a compressing offload accelerator that exploits activation sensitivities and distributions to maximize compression. JPEG-ACT extends compressed offload through the use of domain-specific lossy compression. The key insights exploited by JPEG-ACT are (1) that dense activations are similar to images but with a modified frequency distribution (Figure 2), and (2) that CNNs have error sensitivities that differ from human perception. JPEG-ACT adjusts JPEG compression to optimize for use with CNNs. During the forward pass, JPEG-ACT compresses data before sending it to CPU memory via Direct Memory Access (DMA). During the backward pass, JPEG-ACT decompresses data retrieved from CPU memory before placing it in GPU memory. JPEG-ACT works with both dense and sparse activations and improves training performance versus accuracy loss.

The contributions of this paper are as follows:

- We propose Scaled Fix-point Precision Reduction (SFPR), a method allowing JPEG-ACT to use an 8-bit integer compression pipeline instead of floating-point.
- We optimize JPEG for activation compression of CNNs to account for differing sensitivity to information loss during CNN training versus human perception. This achieves 5.8× (stock JPEG) and 8.5× (optimized JPEG) compression ratio over uncompressed, and 1.98× over the state-of-the-art, GIST [13], with <0.4% change in trained accuracy.
- We propose and evaluate JPEG-ACT, an offload accelerator for JPEG and SFPR, demonstrating a performance improvement of 2.6× over uncompressed offload, and 1.6× over GIST, using <1% GPU area.

We begin by giving an overview of algorithms for training CNNs, and activation compression (Section II), then detail our accelerator design (Section III), and optimization of the JPEG parameters (Section IV). Finally, we report our experimental setup and evaluation of JPEG-ACT (Sections V and VI).

II. Background

This section reviews neural network training, related work on activation compression and the JPEG algorithm.

A. SGD and Backprop

Figure 3 illustrates the training process for a typical contemporary CNN. The backpropagation algorithm [6] has three stages: forward propagation, backward propagation, and update. Forward propagation is performed by starting with an input image, and applying a sequence of layer functions from the first to last layer in the network. The loss (L in Figure 3) is calculated from the final layer's output and it quantifies the error of the network output when compared with the desired or target output. To train the network the gradient of the loss is propagated in the reverse direction by calculating the gradient of the loss with respect to each layer's inputs ($\nabla x \equiv \partial L/\partial x$, Figure 3). Then, the gradient of the loss with respect to each weight ($\nabla w_i \in \nabla w$, bottom, Figure 3) is calculated and updated according to the SGD update:

$$w_i^{t+1} = w_i^t - \eta \nabla w_i^t \tag{1}$$

where t is iteration number and η is a learning rate parameter used to adjust how aggressively weights are updated.

Backpropagation requires that activations be saved after being computed in the forward pass to avoid recomputing them in the backward pass. Most layer's gradients (e.g. conv, norm, ReLU) are calculated using both the input activation and output activation gradient. Recomputation approximately doubles floating-point operations (FLOPs) in the backward pass, which can significantly increase training times.

Gradient calculations can be reformulated to modify which activations need to be saved. The ReLU layer has multiple formulations with similar computation cost. The ReLU forward and backward calculations are, respectively:

$$r = (x > 0) ? x : 0 \tag{2}$$

$$\nabla x = (x > 0) ? \nabla r : 0 \tag{3}$$

From Eqns. 2 and 3, either the input, x, or the output, r, can be used in the backward pass as $(r > 0) = (x > 0)$. Alternatively, a binary mask, $(x > 0)$ can be used instead of x [13] (discussed in Section II-B1).

Frameworks choose which activations to save by examining the overall network structure, minimizing the total computation, and then discarding unused activations. Determining

which activations to store requires information about all network layers, hence dynamic CNN frameworks select on a per-layer basis. Most (Caffe2, Pytorch, and Chainer [22], [23]) use the following strategy: save the conv input, norm input, and ReLU output (r, c and y, resp., Figure 3). These choices are based on knowledge of the computation required to calculate gradients from each activation. For instance, the conv input (r, Figure 3) is required for gradient calculation, and is expensive to recalculate from the output, c. This results in frameworks discarding c if it is not required by another layer's gradient.

We focus on applying compression to activations in the conv/norm/ReLU (CNR) block (Figure 3), used in nearly all modern CNNs. CNR blocks smooth the loss landscape, allowing the training of a wide variety of deep CNNs [3], [9], [10], [24]–[26]. Previously, networks containing alternating conv/ReLU layers only required memoizing the sparse ReLU activation [21]. The introduction of norm ((3), Figure 3), however, adds the requirement that the dense conv output must be saved. Due to this, ReLU compression, such as in [21], covers less than 50% of modern network storage. Although we focus on the CNR block, the compression methods that we use are flexible enough for other sparse and dense activations such as dropout, pooling, and summations.

B. Activation Compression

To compress activations, we require a method that can sustain both a high compression rate and throughput to match the GPU memory system. Compression methods can be classified as either lossless or lossy. Lossless compression algorithms allow the original data to be perfectly reconstructed, whereas lossy compression permits reconstruction of an approximation of the original data. By allowing partial reconstruction, lossy methods discard irrelevant portions of the data to greatly increase the compression rate. Moving between high compression error and high compression rate is commonly called the rate-distortion trade-off. We will now detail prior works on activation compression and the JPEG algorithm for images.

1) Binary ReLU Compression (BRC): Binary ReLU Compression was formulated by Jain et al. [13] to compress ReLU activations to 1-bit. The sign bit of the input ReLU activation is saved, effectively saving $(x > 0)$ instead of x in Eqn. 3. BRC can be used on a ReLU activation provided it is not immediately followed by a conv layer. Networks involving dropout, which include VGG [27] and Wide ResNet [25], can use BRC, but not ResNet [3].

2) Precision Reduction: Many studies focused on inference have explored reducing the precision of activations [28]–[31], however, few examine training [13], [32]–[37]. To the best of our knowledge, most require extensive network modifications [33]–[37], with the exception of Dynamic Precision Reduction (DPR) [13] and Block Floating Point (BFP) [32], [38].

In DPR, 32-bit activations are cast to either 16-bit or 8-bit floating-point values after the forward pass to reduce activation storage, however, Jain et al. noted the difficulty in using 8-bit activations for deeper networks, such as VGG. Jain et al. use this in addition to Compressed Sparse Row (CSR) storage for sparse activations. The authors decreased activation storage by up to 4× using DPR [13].

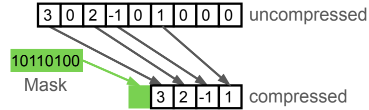

Fig. 4: An example of Zero Value Compression (ZVC) of 8 values

In BFP, fix-point values are used with power-of-two scaling factors for a group of activations [38]. Courbariaux et. al train networks on 10-bit multiplications using BFP [32].

3) Run-length Encoding: ReLU and dropout activations have 50-90% sparsity [21], lending to zero-based compression methods. Run-length encoding [39] has previously been investigated for activation compression and found to give poor results [21]. The method is highly sensitive to sparsity patterns, as it compresses "runs" of zeros. As well, dense activations (e.g. conv) cannot be compressed in this manner.

4) Zero Value Compression (ZVC): Randomly spaced zero values are compressed easily using Zero Value Compression [21], a derivative of Frequent Value Compression [40]. In ZVC, a non-zero mask is created, and the non-zero values are packed together (Figure 4). The mask limits the maximum compression ratio to 8× for 8-bit values. A key advantage of this method is that it works equally well regardless of zero value distribution. The authors achieve a compression ratio of 2.6× on ReLU and dropout activations using ZVC.

5) JPEG: JPEG is a commonly used image compression algorithm [41]. JPEG represents high-frequency spatial information in an image with less precision as this is less important to perception. Below we summarize relevant portions of JPEG. Additional details can be found elsewhere [41]–[43].

Figure 5 illustrates the JPEG algorithm. JPEG splits images into 8×8 blocks of adjacent pixels and quantizes them in the frequency domain. Due to space limitations in Figure 5 we represent these blocks as 3×3 matrices. A block of pixels, represented with integers (❶), is passed through a Discrete Cosine Transform (DCT, ❷), which converts them to the frequency space (❸). Next division quantization (DIV, ❹) is applied. Here frequency values are quantized after dividing them by corresponding entries in the Discrete Quantization Table (DQT, ❺). As the division output is quantized to 8 bits, a high value in the DQT results in fewer bits kept and thus a higher compression. Quantization produces a matrix with a large number of zeros (❻). These are removed in the next stage, Run-length and Huffman coding (RLE, ❼). RLE is lossless and removes zeros by storing run-value pairs. Huffman coding converts these using variable width codes (❽) to produce the final output (❾).

III. The JPEG-ACT Accelerator

JPEG was selected for this work as it was designed for image compression. Convolutions, in essence, are image processing kernels, and it follows that activations resulting from

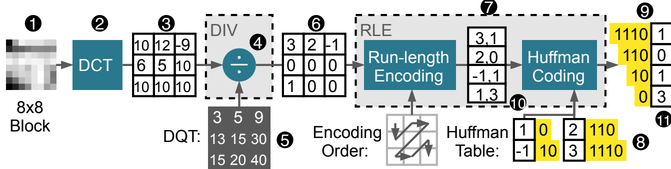

Fig. 5: JPEG encoding example. This illustration uses smaller blocks (3×3 instead of 8×8) due to space limitations.

the convolution of images would also resemble images. To test this hypothesis, we analyze the Shannon information entropy [44] in the spatial (before DCT) and frequency (after DCT) domains (Figure 6). Our experiments demonstrate that the spatial correlations persist deep into the network, even after 40 convolution layers. Frequency domain entropy is lower, especially in the early layers of the network, where activation storage requirements are higher. This implies that the frequency domain is a more compact representation for convolution activations. We do not observe this trend for sparse activations (e.g. ReLUs).

JPEG-ACT is a compressing offload accelerator, similar to cDMA (Figure 1). However, the goals of JPEG-ACT are to address the issues introduced by modern networks. Offloading using cDMA or vDNN has a high overhead due to low PCIe bandwidth and, in networks such as ResNets, due to a low sparsity and/or high proportion of dense activations. GIST avoids the PCIe bottleneck by compressing to GPU memory instead. This removes offload times but uses precious computation resources to perform activation compression. As well, the compression rates provided by GIST ($2.2\times$ - $4.0\times$) result in only moderate relief from activation storage, and still require large amounts of costly GPU memory. JPEG-ACT instead addresses the PCIe bottleneck through an aggressive lossy compression scheme, allowing for a cheaper memory solution, and addresses compute overheads through a custom hardware implementation, avoiding the use of general compute resources.

In this section, we present an overview of the system and JPEG-ACT offload accelerator (Section III-A), how 2D activations map onto the accelerator (Section III-C), and the implementations of the JPEG-ACT components (Sections III-B, and III-D to III-G).

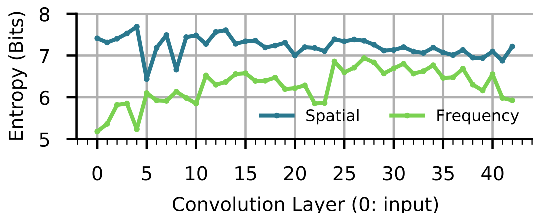

Fig. 6: Convolution activation entropy averaged over all epochs and activations in each layer for ResNet50/CIFAR10.

A. Overview

The baseline system comprises a GPU with High Bandwidth Memory (HBM) and DMA over PCIe to CPU DRAM (Figure 7a) [45]–[47]. We assume each Streaming Multiprocessor (SM), L2 cache/memory controller partition (L2/MC), and DMA unit, are connected using symmetric links to the GPU crossbar. Training using vDNN on this system involves offloading each activation over DMA after its usage in the forward pass (Figure 1). Similarly, in the backward pass, activations are loaded over DMA into GPU HBM before their first use and freed after use. This process is overlapped with compute.

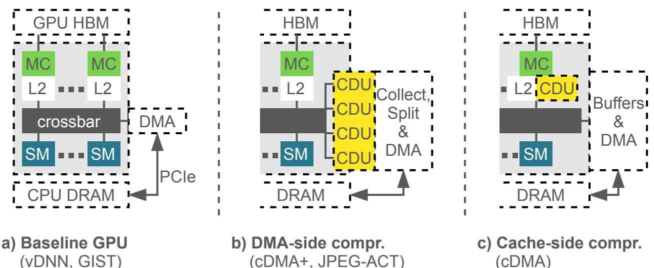

Fig. 7: Compression/Decompression Unit (CDU) locations. a) supports no compression, or software compression. CDUs represent ZVC/ZVD for cDMA or cDMA+, or the JPEG-BASE or JPEG-ACT CDU

For compressed offload, we augment the DMA with several Compression/Decompression Units (CDUs), and a collector/splitter between the CDUs and DMA (Figure 7b). With this system, the maximum effective offload rate is the compression ratio multiplied by the PCIe bandwidth. We use a multi-link, multi-CDU design to avoid being limited by crossbar link bandwidth, explored further in Section VI-E. It is also possible to store compressed data in the GPU HBM, however, we do not investigate this as activations can be as large as 1GB, requiring a large amount of HBM. In this system, compressed traffic from the multiple CDUs is aggregated by the collector when transferring to the CPU, and uncompressed traffic is distributed among CDUs by the splitter when transferring to the GPU.

DMA-side compression (Figure 7b) in this work differs from the cache-side compression (Figure 7c) used by cDMA [21]. Cache-side compression requires a large area and power overhead due to the replication of CDUs across the many cache partitions on modern GPU architectures (e.g., 48 on Volta, [48]). Additionally, for load balancing, sequential cache lines are typically distributed across memory partitions [47]. JPEG compression operates on eight rows of the activation, hence spans up to eight cache lines. This would require inter-cache communication across memory partitions for a cache-side design, thus we perform JPEG exclusively at the DMA-side. We examine locating parallel portions of the CDU at the cache in Section VI-E. For comparison, we re-implement cDMA (Figure 7c) as a DMA-side technique, cDMA+ (Figure

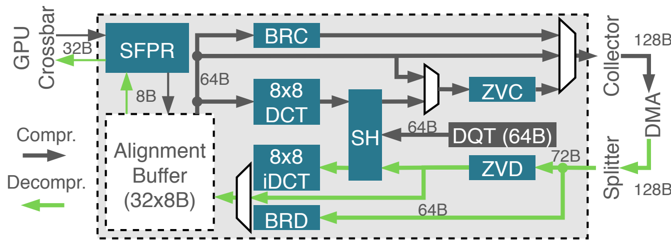

Fig. 8: The JPEG-ACT CDU. SH: Shift unit, BRD: Binary ReLU Decompression, ZVD: Zero Value Decompression

7b). cDMA and cDMA+ have identical CDUs, and differ only in their CDU location and number.

We will start by giving a brief high-level overview of the JPEG-ACT CDU which operates in either compression or decompression mode (Figure 8). Scaled Fix-point Precision Reduction (SFPR) is introduced to convert 32-bit floating-point values to 8-bit integers while keeping quantization error low (Section III-B). SFPR is located between the GPU crossbar and a 256B alignment buffer, allowing four JPEG blocks to be loaded from GPU cache for simultaneous processing (Section III-C). The DCT and iDCT units are pipelined units composed of eight 8-point DCT units to operate on all 64 values in a block at once (Section III-D). The last two stages of the JPEG-ACT compression pipeline are shift quantization (SH) and ZVC/ZVD (Section III-F). In compression mode, the result is sent to the collector, which combines the output from multiple CDUs for sending through the DMA unit (Section III-G). In decompression mode, the compressed input is read from the splitter output (Section III-G).

We find that JPEG standard DQTs (jpeg80 and jpeg60) lead to poor results and that DQT coefficient selection has a large impact on training accuracy. We tune the JPEG DQT for activation compression, by optimizing over the compressed entropy and recovered activation error on ResNet50 (Figure 9). From this procedure, we select optimized low and high compression DQTs, optL, and optH. Finally, we introduce a piece-wise DQT that trains in two stages (optL5H). The selection of these DQTs is discussed in Section IV.

B. Scaled Fix-point Precision Reduction (SFPR)

We propose Scaled Fix-point Precision Reduction (SFPR), a technique to cast from the 32-bit floating-point activations to 8-bit integers to reduce both hardware costs and compression error. Activations are generally represented by floating-point values, however, JPEG compression operates on integers.

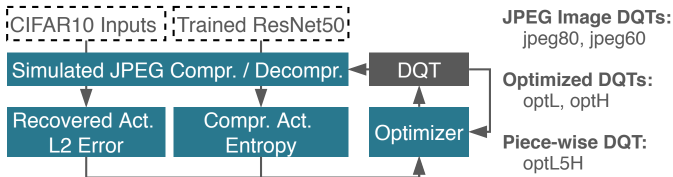

Fig. 9: JPEG DQT optimization procedure.

Naive casting to integers can cause large errors on activations with a dissimilar range to the target integer format. Thus, we develop SFPR to normalize the activation scale, while converting them to integers.

SFPR involves a channel-wise max-scaling of a 4D input activation tensor, $x \in \mathbb{R}^{N \times C \times H \times W}$, followed by clipping to an 8-bit signed integer:

$$s_c = S / \max_{nhw}(|x_{nchw}|) \tag{4}$$

$$y_{nchw} = \text{clip}\left([2^{m-1} s_c x_{nchw}], -2^{m-1}, 2^{m-1} - 1\right) \tag{5}$$

where m is the integer bit width (i.e., 8), [...] denotes the round-to-nearest function, and clip(..., A, B) trims values outside of the range $[A, B]$ to the nearest value within the range. x_{nchw} and y_{nchw} are the original and scaled activations, with n, c, h, w ($< N, C, H, W$) indicating the batch, channel, height, and width index, respectively. The global scaling factor, S, is a hyper-parameter specifying how much of the range of the activation should be clipped to the integer max. The per-channel scaling factor, s_c, is dependent on the maximum of the channel over all batches, $\max_{nhw}$. We compute s_c on a per-layer basis during training.

The SFPR global scaling factor, S is selected by minimizing the recovered activation error, both when compressing with SFPR alone, and when combining it with JPEG with different DQTs (Figure 10). By definition, $S = 1$ results in no activations being clipped. Two effects increase activation error with varying S, clipping and truncation. A high value of S results in high magnitude activations being clipped to the integer min or max ($S \to \infty$, Figure 10). A low value of S results in low magnitude activations being truncated to zero ($S \to 0$, Figure 10). SFPR compression has a low sensitivity to the value of S, with an average increase in recovered activation error of 5.0×10^{-5} across the range $[0.5, 1.25]$. When SFPR is combined with JPEG (SFPR+DCT+...), truncation error ($S \to 0$) increases due to quantization following the DCT. We select a value of $S = 1.125$, which minimizes the overall error of SFPR, JPEG-BASE (...+DIV+RLE), and JPEG-ACT (...+SH+ZVC). A single value of S is used across all networks

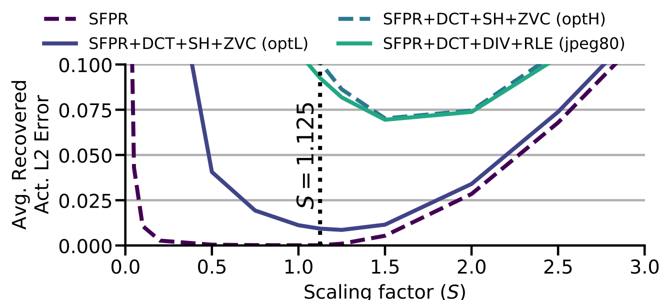

Fig. 10: Scaling factor landscape for ResNet50/CIFAR10 at epoch 5 with conv and sum activations. Each line represents a compression method, with optL, optH, and jpeg80 being DQTs.

and layers to avoid introducing additional hyper-parameters into training.

The channel-wise scaling factor, s_c, could require the costly calculation of the maximum of each channel in the activation map. To avoid this, the maximum can be calculated efficiently using the activation statistics (mean and variance) already determined by batch normalization [24]. Alternatively, prior work on integer quantization has shown that activation statistics do not vary significantly between batches [49], making a sampling-based method a promising approach. We do not measure scaling factor calculation due to the many solutions with little or no performance or hardware overhead.

SFPR, when used as a pre-stage to JPEG, has the benefit of scale normalization. We find that without scale normalization, compression ratios vary greatly during training, and across different networks. Compression variation across channels also reduces trained network accuracy. This appears to result from different input ranges to JPEG compression: activations with a small range will be truncated, giving a high compression ratio but also high compression error. For instance, activations with a range smaller than 1.0 result in zeros after integer casting. Scale normalization ensures that the entire 8-bit integer range is utilized for all activation channels with SFPR and JPEG.

The SFPR compression unit used in the JPEG-BASE and JPEG-ACT accelerator designs is divided into eight identical SFPR Processing Elements (SPE1 to SPE8), each of which handle the conversion of one integer or float (Figure 11).

During the forward pass, s_c is loaded when starting each new channel (❶). Then, after the return of a cache sector (32B) through the GPU crossbar, the eight 32-bit floating-point values on this sector are split among the SPEs (❷). s_c is multiplied with the activation using a 2-stage floating-point multiplier (❸), and cast to an 8-bit integer (❹). Casting of out-of-range values saturates, rather than truncates the resulting value. The results of each SPE are concatenated and saved to the alignment buffer (❺) to await JPEG compression.

During the backward pass, the inverse of the scaling factor ($1/s_c$) is loaded (❶). Inverse scaling factors can be calculated at run time without significant overhead, as the computation cost is amortized for each channel due to the large spatial dimensions of the activations. Eight 8-bit integers (having been decompressed using JPEG) are loaded and split among the SPEs (❻) and converted back to 32-bit floating-point values. The values are multiplied by $1/s_c$ (❸) and concatenated before being sent to the GPU crossbar (❼).

SFPR has some similarities to DPR [13] and BFP [32]. SFPR reduces hardware area versus DPR by converting to 8-bit integers instead of floats. The channel-wise scaling of BFP is similar to SFPR, however, SFPR adds scale normalization, which allows for better utilization of the integer data type.

C. Alignment Buffer

The alignment buffer is a structure designed to convert between the linear address space and the 8×8 blocks ($H\times W$) required by JPEG. As the DCT is a 2D operation, it requires that all 64 elements in a block be available before processing.

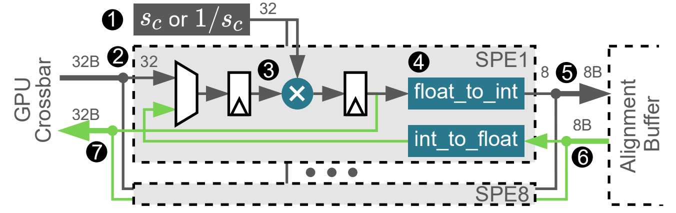

Fig. 11: The SFPR unit showing 8 SFPR Processing Elements (SPEs). In forward mode (grey arrows), $8\times$ 32-bit float values are multiplied by s_c and cast to $8\times$ 8-bit integers, and in backward mode (green arrows), the integers are cast to floats and multiplied by $1/s_c$.

The buffer is sized to hold enough JPEG blocks to prevent duplicate cache line accesses. This requires that activations be padded and aligned such that the start of each cache line coincides with a JPEG block.

The size of the alignment buffer is determined by the JPEG block size, cache line size, activation data type, and SFPR compression ratio. We assume an NCHW memory layout (batch, channel, height, width) for activation tensors, as it has the highest performance for training CNNs [50], and is the default for many frameworks [22], [23], [51]. As each JPEG block has a height of eight elements, a single block can span at most eight cache lines. A single 128B cache line [48] can contain values from up to four JPEG blocks with 32-bit activations. Hence, the alignment buffer is sized to cover eight cache lines compressed to eight bits per activation, i.e. 256B or four JPEG blocks (Figure 12). A smaller buffer would result in duplicate cache line accesses.

The JPEG-ACT CDU requires that blocks are aligned with cache line boundaries. The access stride depends on whether

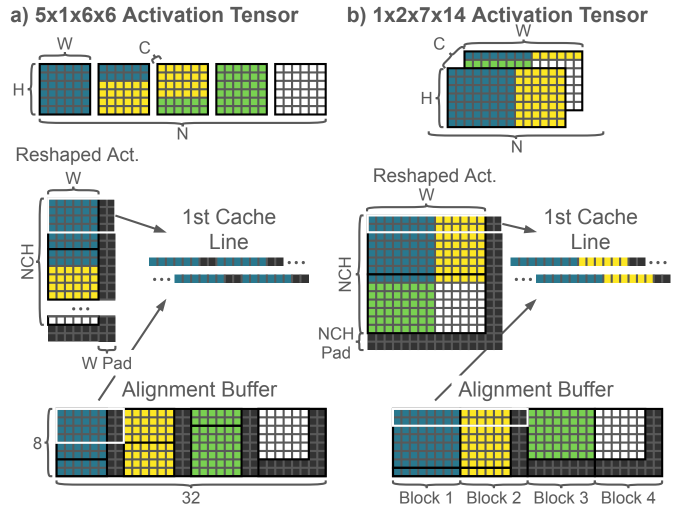

Fig. 12: Memory layout and padding examples. Colors correspond to different 8×8 JPEG blocks and padding. White outline: First cache block boundary, Black outline: Original activation tensor boundary.

the activation tensor has $W \leq 32$. If $W \leq 32$, eight sequential cache lines are loaded, containing exactly four JPEG blocks (Fig. 12a and 12b). If $W > 32$, eight cache lines with a stride of W are loaded (not shown). To force alignment, we zero pad the input activations' width up to a multiple of the JPEG block width, eight elements (W pad, Figure 12a). Rather than padding the height of each activation channel, we instead pad a reshaped activation. The 4D tensors, $\mathbb{R}^{N \times C \times H \times W}$, are reshaped to a 2D tensor, $\mathbb{R}^{NCH \times W}$, and padded along the reshaped dimension (NCH pad, Figure 12b). Reshaping requires no data movement as only the indices are changed, and padding in this manner requires no framework modifications.

Padding increases the memory footprint of the activations and causes a performance overhead, however, this increase is usually small. Padding could be performed at the hardware level, however, this introduces additional hardware and unaligned access overheads. It is preferable to have $N \in 8, 16, 32, \ldots$ due to warp sizing on GPUs, which results in no NCH padding. Similarly, activation tensors with $W \in 8, 16, 32, \ldots$ result in no W padding. Out of the datasets and networks this work examines, only ResNet18/ImageNet and ResNet50/ImageNet [3] require padding, with a storage overhead of 6.4% for H, W padding, and 3.0% for NCH, W padding on ResNet50. These overheads are low as most activation storage is in the widest layers of the network, making the relative size of the padded elements small.

The alignment buffer is designed with one 8B read/write port, and one 64B read/write port. During compression, the SFPR unit may perform 8B writes, while the DCT and other compression units perform 64B reads. Once the first JPEG block has been loaded, the DCT stage proceeds until all 4 blocks have been read (4 cycles) and the buffer is freed for use by the next set of blocks. During decompression, roles are reversed, with 64B writes from the decompression pipeline, and 8B reads by the SFPR unit. Structuring allows us to maintain fewer read and write ports on the buffer.

D. Discrete Cosine Transform

The DCT unit used by JPEG-BASE and JPEG-ACT is implemented by utilizing eight 8-point 1D DCT units (Figure 13). We use the well known 8-point DCT of Loeffler et al. (the LLM DCT) [52] due to its ease of pipelining, and efficient use of multipliers. The LLM implementation requires 11 multiplications, and 29 additions for each 8-point DCT, resulting in 88 multipliers for the JPEG-ACT DCT. We implement the JPEG-ACT DCT as two passes through the 1D DCT units. After computing the DCT along the first dimension, the block is transposed and processed again for the DCT along the second dimension. Each pass through the unit takes four cycles to complete. After being transformed by the 2D DCT, the block is sent to the DIV unit for JPEG-BASE (not shown) or the SH unit for JPEG-ACT (right, Figure 13 and Section III-F).

The iDCT unit is fashioned similarly to the DCT unit. In brief, eight 8-point iDCT units are combined with a normalizing shift stage. The stages are inverted relative to the DCT, i.e. multipliers become dividers, etc. This results in a similar

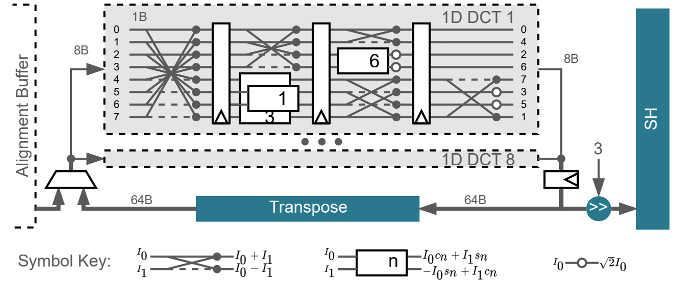

Fig. 13: The JPEG-ACT 2D DCT unit. 1D DCT is reproduced from the LLM fast DCT [52]. Bottom: DCT algorithm building blocks with $c_n = \cos(n\pi/16)$, $s_n = \sin(n\pi/16)$.

implementation with a two-pass structure and four pipeline stages.

E. DIV and RLE (JPEG-BASE)

JPEG-BASE uses a hardware implementation of the JPEG standard quantization and coding stages. DIV quantization is a simple division by the DQT, and RLE coding combines run-length encoding and Huffman coding. We implement the DIV unit as a parallel multiplier, and use designs from OpenCores for RLE (encoding) [53], and RLD (decoding) [54]. Hardware is duplicated as necessary to meet throughput requirements.

F. SH and ZVC (JPEG-ACT)

The JPEG standard was designed for software compression of images. We developed the shift (SH) and ZVC back-end, replacing steps from the standard JPEG algorithm to reduce hardware overheads and improve compression on activations. SH quantization is designed to remove the multipliers used in the DIV stage of JPEG-BASE. ZVC coding [21] is used because of our observations that activation frequency distributions vary drastically from images. SH and ZVC, combined with SFPR and the DCT, compose the JPEG-ACT accelerator.

SH is motivated by our observations that exact quantization is often unnecessary. By switching the division to a shifting operation (Figure 14), the area associated with the quantization operation can be reduced by 88%. This has the effect of limiting DQT values to powers of 2. In compression mode, 64 right shift operations are performed in parallel. In decompression mode, the right shifts are replaced by left shifts. SH comes at the expense of having only eight available quantization modes for each frequency. When performing activation compression,

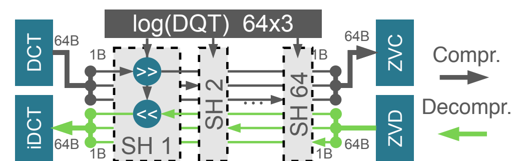

Fig. 14: Shift (SH) unit showing 64 parallel units. The log DQT has 64, 3-bit outputs. Colored arrows indicate compression and decompression paths.

we observe that fewer quantization modes are sufficient as the individual effect of single frequencies is reduced (Section IV).

We use ZVC to compress the sparse result of the SH stage. After the DCT and quantization, images have most zeros at high frequency values (Figure 2). Conversely, activations display a flatter profile, with zeros randomly distributed across mid and high frequencies (Figure 2). Thus, ZVC has a higher compression than RLE on frequency domain activations.

The modifications of SH and ZVC decrease hardware area by 1.5× (Section VI-F), and increase compression by up to 1.4× (Section VI-C).

G. Collector and Splitter

The collector and splitter units are required to convert between the multiple CDU data streams and the single PCIe DMA data stream. The collector joins the variable-sized streams from the CDUs into a single stream. The splitter splits the PCIe stream by calculating and tracking the number of bytes in each block. Both the collector and splitter connect directly to the PCIe DMA unit.

The scheduling policy for interleaving CDUs can have a large impact on collector and splitter designs, hence it needs to be addressed first. The collector and splitter operate at a rate of one 8×8 block per cycle (Figure 15). The load or store rate to the GPU crossbar is one block per eight cycles per CDU. Hence, the entire JPEG-ACT accelerator will always be bottlenecked by either the PCIe interconnect at low compression rates or the crossbar link at high compression rates. As the CDU processing is 8× faster than the crossbar rate, we use a simple round-robin scheduling of the CDUs accomplished with a simple mux (Figure 15), i.e. CDUs are scheduled in order with one cycle each. This also solves the issue of splitting, as streams are deterministically interleaved.

The collector unit operates during the forward pass (Figure 15a). One CDU writes to the collector on each cycle with a round-robin policy (❶). The ZVC mask is summed to obtain the total number of non-zero bytes in the block (❷). The primary structure for aligning non-zero values is the 256B Input FIFO (IFIFO, ❸). The IFIFO is designed to allow a variable-sized push operation from 0B to 72B, indexed by the push_bytes signal. When the IFIFO fill is greater than 128B, 128B is popped from the head of the IFIFO and the filled packet is sent to the DMA unit (❹). As pop operations are always 128B, the IFIFO tail location is always at the 0th or 128th byte.

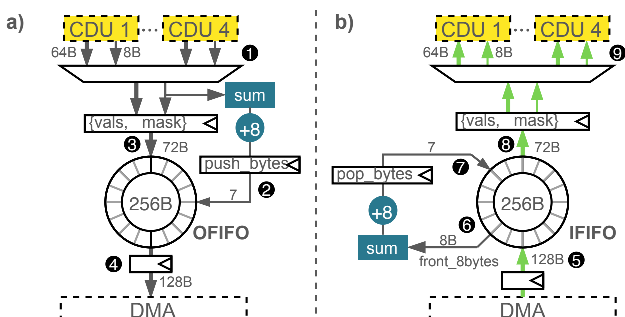

Fig. 15: a) Collector and b) Splitter units for aggregating and splitting compressed streams. Pipeline registers and control signals inserted as necessary.

The splitter unit operates during the backward pass (Figure 15b). 128B packets from the DMA are pushed onto a 256B Output FIFO (OFIFO, ❺). Eight bytes, representing the mask of the next block to be read, are peeked from the front of the OFIFO (❻). The mask is used to calculate the number of bytes to pop from the OFIFO in the next cycle (❼ and ❽). As collection is deterministic, the distribution of blocks from the splitter occurs with the same round-robin policy (❾).

By utilizing a collector and splitter, multiple CDUs can be used while avoiding issues with inter-cache communication.

IV. Optimizing Compression

The JPEG DQTs for images (jpeg60, jpeg80, etc.) were created by extensively studying human perception, however, prior work indicates that CNNs have a different frequency sensitivity [55]. Optimization is performed by first defining metrics for approximating network convergence and compression, and the creation of an objective function (Figure 9). This results in a significantly higher activation compression rate with similar error relative to a JPEG DQT for images.

Network convergence is currently a poorly understood topic [56]. However, an efficient way of measuring convergence is required to optimize the JPEG DQT. There are no objective functions to gauge the final accuracy of a network without training. Therefore, we choose to maintain accuracy, rather than attempting to maximize accuracy.

The effect of JPEG compression on training can be understood by considering a single layer of a network during training, with reshaped and padded activations, $x \in \mathbb{R}^{NCH \times W}$, and weights, w. For one iteration of backprop and no compression, output activation and weight gradient are calculated as $y = w \circ x$, and $\nabla w = \nabla y \circ x$, respectively, where ∇y is the output activation gradient, and $\circ$ is a generic tensor dot product. If the iteration is repeated using JPEG activation compression, the approximate weight gradient, ∇w^*, is calculated as:

$$q = \left(q_{ij}\right) = \left(\,[\mathrm{DCT}(x)_{ij}/\mathrm{DQT}_{uv}]\,\right) \tag{6}$$

$$x^* = \mathrm{iDCT}(\,\left(q_{ij}\mathrm{DQT}_{uv}\right)\,) \tag{7}$$

$$\nabla w^* = \nabla y \circ x^* \tag{8}$$

where $u, v \equiv i \bmod 8, j \bmod 8$, $[...]$ is the round-to-nearest function, $q \in \mathbb{Z}^{NCH \times W}$ is the quantized frequency matrix, and x^* is the recovered activation.

The tensor dot product is a linear operation, hence the error relative to uncompressed can be expressed as:

$$\nabla w^* - \nabla w = \nabla y \circ (x^* - x) \tag{9}$$

Identical convergence to uncompressed is achieved as the error approaches zero. This can be accomplished by minimizing the L2 activation error, using Eqn. 9 and a first order approximation: $\|\nabla w - \nabla w^*\| \propto \|x - x^*\|$.

To form the global objective function, a measure of compression is also required for the optimization procedure. We use the Shannon entropy (H, Eqn. 11) of the quantized frequency coefficients (q), which represents the minimum bits required per activation. This, combined with the average L2 error per activation (L_2), form the objective function, $\mathcal{O}$:

$$L_2 = (NCHW)^{-1}\|x - x^*\| \tag{10}$$

$$H = \sum_{v=-2^{m-1}}^{2^{m-1}-1} P(q{=}v)\log_2(P(q{=}v)) \tag{11}$$

$$\mathcal{O} = (1-\alpha)\lambda_1 H + \alpha\lambda_2 L_2 \tag{12}$$

where $m = 8$ is the quantization bit width, $P(q = v)$ is the probability that $q = v$, determined by counting the number of occurrences of v in q, and $\lambda_1 = 10$ and $\lambda_2 = 10000$ are normalizing scaling factors. α is a hyper-parameter that controls the rate/distortion trade-off.

We minimize $\mathcal{O}$ with respect to the DQT for all convolution layers using 240 example activations from a generator network with frozen weights, ResNet50/CIFAR10 trained for 5 epochs. The example activations are used to calculate L_2, H, and $\mathcal{O}$ for a given DQT. SGD is used as an optimizer ($lr = 2.0$, $p = 0$) with DQT gradients calculated using forward finite difference (difference of 5). The first of the 64 DQT parameters, representing the activation mean, is fixed to 8 to prevent instability in the batch normalization parameters.

We examine the rate/distortion trade-off for SFPR and different JPEG DQTs to determine the efficacy of optimization (Figure 16). Optimizing the DQTs for activation compression results in lower error for the same compression than both SFPR and JPEG-BASE with image DQTs, and decreases entropy by 1 bit for the same error compared to image DQTs (optH vs. jpeg80, Figure 16).

Tuning of the DQT for the desired compression rate and error is controlled using α, hence we select two values representing low and high compression variants, optL ($\alpha = 0.025$) and optH ($\alpha = 0.005$), respectively. As α increases, a higher cost is placed on L2 activation error, resulting in the error decreasing from 0.10 to 0.02 with optH vs. optL (Figure 16).

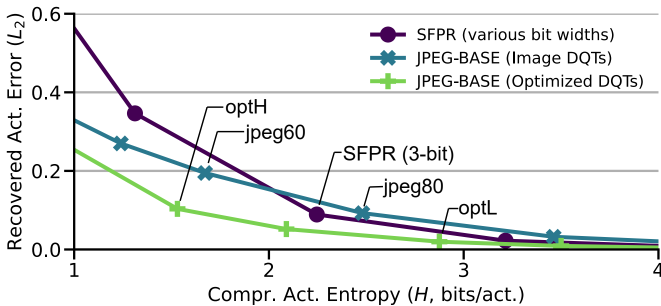

Fig. 16: Rate/distortion trade-off for SFPR (2-, 3-, and 4-bit), and JPEG-BASE with image DQTs (jpeg40, 60, 80, and 90) and optimized DQTs ($\alpha = 0.001, 0.005, 0.01$, and 0.025). Based on ResNet50/CIFAR10 trained for 5 epochs.

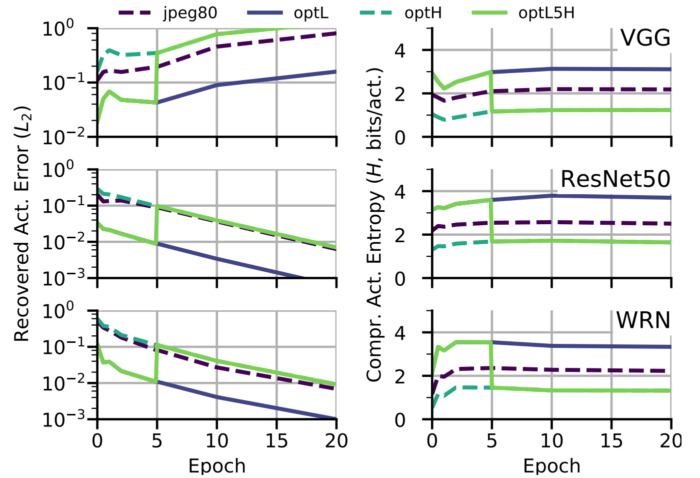

Fig. 17: Activation error and entropy for JPEG-BASE with various DQTs on CIFAR10.

The values of α for optH and optL were chosen as they have a similar error to the jpeg80 and jpeg90 DQTs. This error range was observed to be approximately where a decrease in accuracy begins.

We examine how compression error and entropy vary over the course of training by evaluating each DQT on snapshots of the networks at different epochs (Figure 17). Activation error is highest in the first epochs for ResNet50 and WRN (Figure 17, left), which is a consequence of weight decay. However, we observe that after the first 5 epochs, compression remains constant. This is attributed to stable activation distributions from batch normalization [24], combined with the scale normalization of SFPR. We observe that these trends in error and entropy continue for the remainder of training.

The first epochs of training are critically important to the convergence of CNNs [57]. To address the critical first epochs, we propose a piece-wise approach to selecting DQTs, optL5H (Figure 17). optL5H uses the optL DQT for the first 5 epochs of training, then switches to the optH DQT for the remainder of training. This avoids high errors in the critical period.

V. Methodology

We compress the activations in each CNN according to layer type and dimensions (Table II). The use of BRC is determined by whether a ReLU activation is followed by a conv layer, hence they are divided by subsequent layer. Sum refers to dense activations produced by the addition of two activations. JPEG compression is used on conv and sum activations with size ≥ 8, due to the 8×8 block size of the algorithm. We do not use JPEG on the final four convolutions, or fully connected layers, due to their small activation size.

Datasets and networks are selected from a variety of network types and CNN applications. Extremely large networks, e.g. GPIPE, are not examined in this work due to high memory requirements [9]. We evaluate JPEG-ACT using the CIFAR10 [58], ImageNet [7], and Div2k [59] datasets. We use six image classification CNNs: VGG-16 (VGG) [27],

TABLE I: Compression rate trade-offs. Compression ratios are bracketed, bolded values indicate highest for lossy methods.

	Baseline	cDMA+	GIST 8-bit	SFPR 8-bit	JPEG-BASE		JPEG-ACT		
					jpeg80	jpeg60	optL	optH	optL5H
CIFAR10	*% Top-1 Val. Accuracy (Compression ratio)*								
VGG	92.1	- (1.5x)	92.7 (6.1x)	92.0 (4x)	91.4 (7.4x)	89.0 (8.3x)	**92.8** (9.4x)	91.9 (**12.0x**)	92.4 (11.9x)
ResNet50	94.5	- (1.1x)	94.4 (4.1x)	**94.5** (4x)	93.6 (5.1x)	93.0 (6.0x)	94.4 (5.2x)	93.8 (**7.6x**)	94.4 (7.5x)
ResNet101	94.7	- (1.1x)	94.4 (4.1x)	94.7 (4x)	94.0 (5.0x)	92.8 (5.8x)	**94.8** (5.0x)	94.0 (**7.2x**)	94.5 (**7.2x**)
WRN	95.4	- (1.6x)	**95.8** (5.6x)	95.2 (4x)	92.6 (6.2x)*	91.9 (7.2x)*	95.7 (8.1x)	91.8 (**11.0x**)*	94.2 (10.9x)
ImageNet	*% Top-1 Val. Accuracy (Compression ratio)*								
ResNet18	67.8	- (1.2x)	66.9 (3.6x)	**67.9** (4x)	67.4 (5.7x)	66.6 (6.4x)	67.6 (6.1x)	66.9 (**7.3x**)	67.3 (7.2x)
ResNet50	71.7	- (1.2x)	68.5 (3.7x)	71.4 (4x)	**71.8** (5.3x)	69.8 (**6.1x**)	**71.8** (5.1x)	28.9 (6.0x)*	71.6 (5.9x)
Div2K	*Val. PSNR (Compression ratio)*								
VDSR	35.6	- (1.3x)	34.8 (4.0x)	**35.5** (4x)	**35.5** (5.9x)	35.3 (6.4x)	**35.5** (8.2x)	35.4 (**9.2x**)	35.4 (9.1x)
Average	*% Change from Baseline (Compression ratio)*								
All Models	-	0 (1.3x)	-1.07 (4.5x)	-0.12 (4x)	-0.87 (5.8x)	-2.27 (6.6x)	**+0.07** (6.7x)	-9.58 (**8.6x**)	-0.38 (8.5x)

* run failed to converge

TABLE II: Compression selection by activation type. SD=SFPR+DCT

Method	conv or sum	ReLU (to other)	ReLU (to conv)	pool or dropout
cDMA+	None	ZVC		
GIST	DPR	BRC	DPR+CSR	
SFPR	SFPR			
JPEG-BASE	SD+DIV+RLE *	BRC	SFPR	
JPEG-ACT	SD+SH+ZVC *	BRC	SFPR+ZVC	

* for $NCH, W \geq 8, 8$, otherwise SFPR.

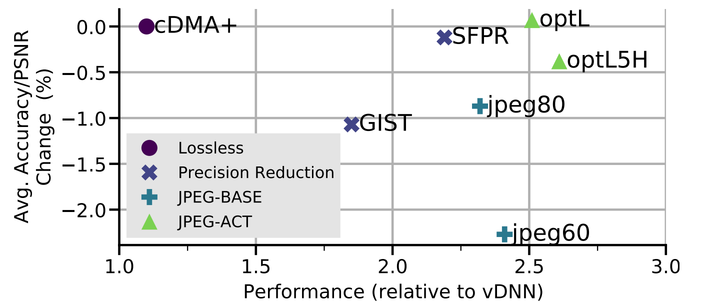

Fig. 18: Percentage accuracy loss vs. relative speedup.

Wide ResNet (WRN) [25], and ResNet18, 50, and 101 [3]. Networks are unmodified from the original sources [22], [60]. Additionally, we examine JPEG-ACT on super-resolution with VDSR [61], which is modified to use 64×64 random crops and batch normalization.

We implement a functional simulation of each method in Chainer [22] to examine compression and its effects on trained neural network accuracy. The methods are implemented as CUDA code that extends the framework. We skip lossless compression during functional simulation, instead calculating compression ratios offline with a batch size of 8.

Performance simulation uses GPGPU-Sim [45], [46], configured to simulate an NVIDIA Titan V GPU [47], and PCIe 3.0 with an effective transfer rate of 12.8GB/s (Figure 7a) [19]. We model boost clocks of 1455MHz, 40 Streaming Multiprocessors, an interconnect capable of 32B/cycle bi-directional bandwidth, and 850MHz HBM. Whole-network performance is assessed by a microbenchmark, programmed in C++, CUDA, cuDNN, cuSPARSE, of three CNR blocks sampled from each network at a batch size of 16, as full networks lead to prohibitive simulation requirements. A warm-up of one ReLU is used to avoid cold start misses in the GPU cache. As source code for GIST is not publicly available we reimplemented it both for performance (CUDA and cuSparse) and functional (Chainer and CUDA) simulation. This includes the DPR, BRC, in-place optimizations and Sparse Storage Dense Compute, a Compressed Sparse Row (CSR) variant.

We implement the JPEG-ACT accelerator as RTL and synthesize using Synopsys Design Compiler to evaluate timing, area, and power requirements. Our synthesis targets the interconnect clock frequency, and 45nm technology using the FreePDK45 design library [62]. Results are scaled to 15nm, as the 15nm library is no longer available, and 50% wire overhead added in a similar manner to prior works [21], [63].

VI. Evaluation

A. Overall

Figure 18 plots percentage change in accuracy versus performance improvement. The two JPEG-ACT variants, optL and optL5H achieve better performance gains for a given level of accuracy loss versus the alternatives considered in this study.

B. Compression and Accuracy

We train all networks under compression and report the best validation score, i.e. the Top-1 accuracy or Peak Signal-to-Noise Ratio (PSNR), and average network compression ratio (Table I). ImageNet accuracies are lower (-4.2%) than the original work [3], [64] as we use a more CPU-efficient augmentation procedure and report the 1-crop validation instead of the 10-crop test accuracy.

cDMA+ is lossless, resulting in no accuracy change, however, it has a low compression ratio of 1.3×. We observe ReLU and dropout compression ratios of 2.1× and 3.9×, similar to

those of Rhu et al. [21]. Networks with batch normalization have up to 60% dense activations, leading to the low overall compression (Figure 19).

Training with GIST results in a significant decrease in accuracy/PSNR when compared to SFPR (-1.07× vs. -0.12×), predominantly in ResNets and VDSR. Jain et al. also observed this issue with VGG/ImageNet and 8-bit DPR [13]. We hypothesize this is due to the truncation of small valued channels. We observe that the minimum per-channel range of activations in these networks is 0.16. With this range, 15% of the 256 available 8-bit DPR values are utilized, while 66% are utilized for SFPR. This could be avoided at the expense of 2× lower compression by using 16-bit GIST [13]. SFPR generally has a higher integer utilization than DPR due to scale normalization, resulting in lower activation error and better accuracy.

GIST compression ratios are significantly higher on networks that contain dropout (VGG, Figure 19, and WRN) versus those that do not (ResNets, Figure 19, and VDSR). The CSR method used by GIST first compresses using 8-bit DPR, then extracts non-zero values and their column index. With the optimizations made by Jain et al., this requires the storage of an 8-bit DPR value and an 8-bit column index per non-zero value [13]. When sparsity is <50%, size increases over DPR alone, which is observed for ResNets on ImageNet (Table I), making CSR a poor choice for networks without dropout. CSR is advantageous when the compressed values are larger than the indices, i.e. with 16-bit DPR.

JPEG-BASE provides improved compression over cDMA+, GIST and SFPR, however, a lower accuracy than SFPR. WRN is most sensitive to lossy compression, as it does not converge with jpeg80. This non-convergence (* in Table I) is observed as a sudden decrease in accuracy during training, which can be used as a warning sign that the compression is too high. For jpeg80, this is only observed with WRN, and with an average accuracy change of -0.54% across the remaining networks. Although jpeg60 provides high compression ratios, the decrease in accuracy for it and lower quality settings (e.g., jpeg40) is too severe to warrant use.

The modifications to create JPEG-ACT provide both a significant increase in accuracy/PSNR and compression. The optimization procedure of optL reduces activation error to obtain a similar or better accuracy than SFPR, and the baseline (Table I). Small decreases in error are likely due to CNN training being a stochastic process. optH causes divergence for WRN/CIFAR10 and ResNet50/ImageNet, however, this is not observed with the piece-wise technique, optL5H. Annealing the networks for the first 5 epochs using optL, then switching to optH, provides an average compression ratio of 8.5× while keeping the accuracy change at -0.38%, less than half that of GIST. JPEG-ACT with optL5H increases compression over JPEG-BASE with jpeg80 both by having a higher quantization from the DQT, and by using ZVC on sparse activations. Higher quantization has a larger effect on sum activations, with conv activation compression remaining mostly unchanged (Figure 19). ZVC can compress ReLU and dropout activations to further decrease their size after SFPR, with a relative

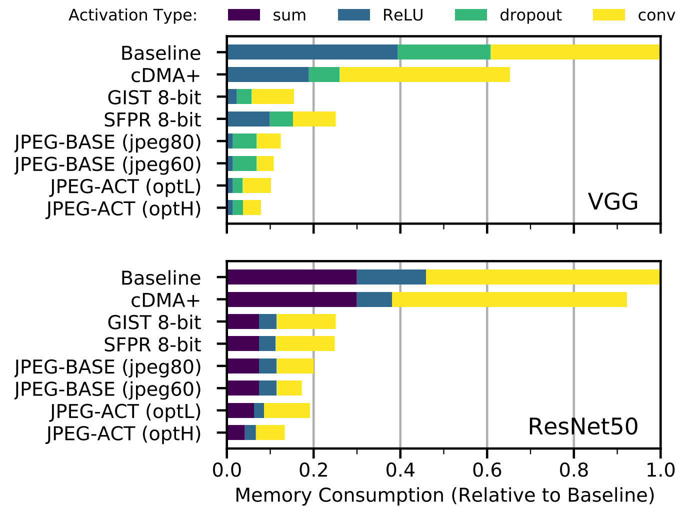

Fig. 19: Activation footprint breakdown by activation type for CIFAR10 models.

contribution of 1.1× to total compression (Figure 19).

Networks that do not converge were examined and found to have diverging activation distributions, i.e. a diverging mean or standard deviation over training. We suspect that activation compression is affecting the activation mean, causing divergence of the mean-dependent batch normalization parameters. Decreasing the compression of the first DQT coefficient, relating to the mean, can reduce this behavior. Similarly, annealing for the first 5 epochs with lower compression (optL5H, Table I) also prevents divergence, implying that this is made worse by rapid changes early in training. More investigation into the training dynamics of CNNs under error is required to fully understand this issue.

C. Quantization and Coding Modifications

To isolate the effects of DQT optimization, quantization, and coding, we evaluate each DQT with each JPEG back end and measure the conv and sum compression ratio (Table III). The jpeg80 DQT has a significantly lower compression than optH, highlighting the effectiveness of the optimization procedure to increase compression (Table III). However, this high compression is at the expense of accuracy (Table I). Training with optL5H results in a compression ratio similar to optH while maintaining accuracies similar to optL (Tables I and III). Using optL5H over jpeg80 increases conv and sum compression by >1.38× for any back end.

TABLE III: ResNet50/CIFAR10 conv+sum compression for various DQTs (top) and JPEG back ends (left)

	jpeg80	**jpeg60**	**optL**	**optH**	**optL5H**
DIV+RLE	5.29	6.43	3.52	7.79	7.72
SH+RLE	5.26	6.32	3.99	7.43	7.38
DIV+ZVC	5.80	6.52	4.62	8.31	8.24
SH+ZVC	5.77	6.46	5.08	8.01	7.96

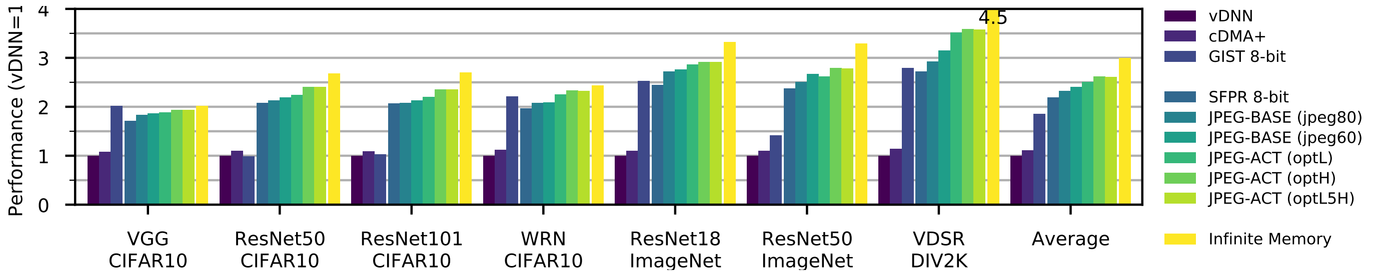

Fig. 20: Relative performance to vDNN.

The use of ZVC over RLE increases the compression ratio by 1.12×. In JPEG-BASE, RLE is used because high-frequency information has a low magnitude, leading to most high-frequency values being zero after quantization. CNN activations, however, have much larger high-frequency modes, which are quantized to non-zero values. RLE performs poorly with randomly distributed zeros in contrast to ZVC. Additionally, the optimized DQTs have a flatter quantization profile when compared to image DQTs. This low-frequency quantization further randomizes zeros and is especially apparent in the improvement of optL when using ZVC (1.3×).

D. Performance

Performance measurement is accomplished through microbenchmarking using CNR blocks with an optional dropout or pooling layer. Due to simulation time and memory constraints, we simulate three layers of each network (the first, middle, and last), and use a batch size of 16. The algorithms used are WINOGRAD, and WINOGRAD_NONFUSED for 3×3 convolutions, and IMPLICIT_GEMM and ALGO_0 for 1×1 convolutions. This is representative of software frameworks such as Chainer and Pytorch [46].

GIST performance is strongly influenced by network structure (Figure 20). Poor performance on ResNet50 and ResNet101 can be attributed to the presence of bottleneck layers [3], which involve 1×1 convolution to decrease the number of channels. Bottlenecks involve up to 2048 channels, creating large activations with 9× fewer FLOPs than a similarly sized 3×3 kernel. The non-zero scan in the cuSparse dense2CSR conversion takes longer than a 1×1 kernel, in this case, creating a large performance overhead.

By comparison, SFPR and JPEG-ACT display performance that is not network dependent. The SFPR-only design provides 1.35× performance over GIST despite having a lower compression, primarily because CSR is slower than SFPR. The PCIe bandwidth limitations are nearly eliminated by JPEG-ACT with optL5H, giving a performance increase over GIST of 1.59× and overhead of 1.13×. More consistent performance is obtained by shifting the bottleneck to effective offload rate instead of compression throughput. Compression increases that result from modifying JPEG for CNNs, improve performance by 1.12× while decreasing the error change from baseline by 2.3× (JPEG-ACT optL5H vs. JPEG-BASE jpeg80). We observe that the remaining overheads of JPEG-ACT are caused by congestion on the GPU interconnect from the increase in DMA traffic. Despite this, JPEG-ACT obtains 2.61× performance versus vDNN.

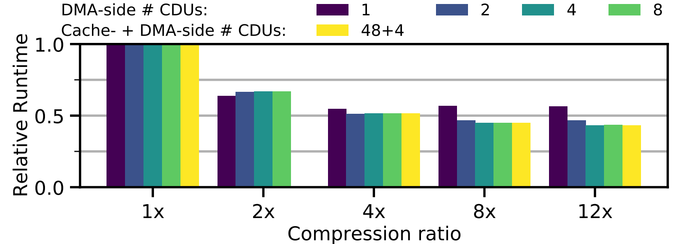

Fig. 21: Performance when changing the number of CDUs on ResNet50/CIFAR10 with a fixed compression ratio. Cache- + DMA-side refers to Cache-size SFPR CDUs, and DMA-side DCT+SH+ZVC CDUs.

VDSR has 1.4× to 2.3× worse offload performance than the other networks (Figure 20). VDSR has no dropout, pooling, or bottleneck layers, however, the most important difference is that all activations have few channels and a large spatial dimension. We have observed that cuDNN launches a different set of compute kernels for VDSR, and hypothesize that the method used has a lower compute density, resulting in poor offload performance.

E. CDU Count and Location

The effective offload rate available to JPEG-ACT is highly dependent on the location and configuration of the CDUs in the GPU memory system. Most importantly, the number of CDUs affects the available bandwidth into the GPU (Figure 21). With DMA-side compression, there is little increase in performance over 1 CDU at 2× and 4× compression as the offload is bottlenecked by the PCIe offload rate. At 8× and 12× compression, however, the bottleneck is removed, and performance increases as CDUs are added. Performance for 12× compression increases by 1.08× when moving from 2 to 4 CDUs, but by less than 0.5% when moving from 4 to 8 CDUs. At this compression and number of CDUs, the memory partitions become the bottleneck, preventing further increases.

We also examine the impact of moving the SFPR portion of the CDU to the L2 cache in a combined Cache- and DMA-side compression (Figure 21). In this configuration in the forward pass, values from the cache are immediately compressed by

TABLE IV: JPEG-ACT synthesis by component

Component	Area (um^2)	Power (mW)
SFPR	44924	34.3
DCT + iDCT	229118	273.4
Quantize (DIV)	12507	14.4
Quantize (SH)	1593	2.5
Coding (RLE + RLD)	125890	176.0
Coding (ZVC + ZVD)	21519	17.1
Collector + Splitter	173445	170.3
Crossbar (+3 ports)	2253427	1668.0

TABLE V: Designs comparison with buffers and 4 CDUs. Crossbar excluded.

	cDMA+	SFPR	JPEG-BASE (jpeg80)	JPEG-ACT (optL5H)
Power (W)	0.26	0.35	1.82	1.36
Area (mm^2)	0.35	0.31	2.16	1.48
Compression	1.3x	4.0x	5.8x	8.5x
Offload (GB/s)	15.6	48.0	69.6	108.8

SFPR, sent over the GPU interconnect, and compressed again by JPEG before the DMA unit. The minimum compression rate is 4× due to the mandatory use of SFPR. As there is one SFPR unit per memory partition, there are 48 SFPR CDUs and 4 JPEG CDUs. This configuration has a high area overhead due to duplication of the SFPR units and results in a performance increase of 1% over a 4 CDU DMA-side design.

F. Synthesis

Power and area results for the individual JPEG-ACT components (Table IV), indicate that the DCT is the most expensive component of JPEG-ACT, followed by the required buffers. The overall area and power for each design are visible in Table V. When compared to cDMA+, JPEG-ACT provides a significant increase in effective PCIe bandwidth while maintaining an area and power <1% of an NVIDIA Titan V GPU. This is even smaller relative to larger data center GPUs [65]. The modifications to the JPEG-ACT back end for CNNs reduce overall area and power by 1.3× and 1.5× , respectively, while increasing available PCIe offload bandwidth.

VII. Related Work

We compare favorably against the primary works examining activation storage during training, i.e. vDNN [19], cDMA [21], and GIST [13]. However, there are many proposals for compressing pre-trained neural networks to reduce costs at inference , which, unlike JPEG-ACT, do not decrease activation storage during training [33]–[37], [66]–[70]. These methods include frequency transforms [68], [69] and precision reduction [33]–[37], [70]. Training networks with a reduced precision (e.g. 1, 3, or 8 bits), while effective, requires modification of the CNN framework, network architecture, and training schedule [33]–[37], which is not necessary with JPEG-ACT. Other works have examined reduced precision gradient storage for multi-GPU training, which does not decrease local memory consumption [71].

Stored activations can be removed entirely, either through removing gradients [72] or by using reversible networks [12]. These methods involve a much higher computational load than more conventional compression methods, as gradients [72] or convolutional activations [12] need to be regenerated. In contrast to JPEG-ACT, this restricts the available layer types of the network.

VIII. Conclusion

We have presented JPEG-ACT, a novel offload accelerator for CNN activation compression, and its fixed-point compression mechanism Scaled Fix-point Precision Reduction (SFPR). Our results demonstrate JPEG-ACT can be effectively used on a wide variety of datasets and benchmarks, and provides significantly higher compression ratios than the state-of-the-art. JPEG can be further tuned for CNNs, providing a 1.5× improvement in compression, while increasing trained accuracy. Given hardware support, JPEG-ACT can be incorporated simply with any CNN architecture and framework

References

[1] A. Krizhevsky, "Convolutional Neural Networks for Object Classification in CUDA," University of Toronto, EECE1742S: Programming Massively Parallel Multiprocessors Using CUDA, April 2009.

[2] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," in *Proc. Int. Conf. on Neural Information Processing Systems (NeurIPS)*, 2012.

[3] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," in *Proc. IEEE/CVF Conf. on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 770–778.

[4] T.-Y. Lin, P. Dollr, R. Girshick, K. He, B. Hariharan, and S. Belongie, "Feature Pyramid Networks for Object Detection," *arXiv:1612.03144 [cs]*, 2016.

[5] J. Yao, S. Fidler, and R. Urtasun, "Describing the scene as a whole: Joint object detection, scene classification and semantic segmentation," in *CVPR*, 2012, pp. 702–709.

[6] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning internal representations by error-propagation," in *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, 1986.

[7] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "ImageNet: A Large-Scale Hierarchical Image Database," in *CVPR*, 2009.

[8] E. Real, A. Aggarwal, Y. Huang, and Q. V. Le, "Regularized Evolution for Image Classifier Architecture Search," in *Proc. AAAI Conf. on Artificial Intelligence*, 2019, pp. 4780–4789.

[9] Y. Huang *et al.*, "GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism," in *NeurIPS*, 2019, pp. 103–112.

[10] M. Tan and Q. Le, "EfficientNet: Rethinking model scaling for convolutional neural networks," in *Proc. Int. Conf. on Machine Learning (ICML)*, 2019, pp. 6105–6114.

[11] T. Chen, B. Xu, C. Zhang, and C. Guestrin, "Training Deep Nets with Sublinear Memory Cost," *arXiv:1604.06174v2 [cs]*, 2016.

[12] A. N. Gomez, M. Ren, R. Urtasun, and R. B. Grosse, "The Reversible Residual Network: Backpropagation Without Storing Activations," in *NeurIPS*, 2017, pp. 2214–2224.

[13] A. Jain, A. Phanishayee, J. Mars, L. Tang, and G. Pekhimenko, "Gist: Efficient Data Encoding for Deep Neural Network Training," in *Proc. ACM/IEEE Int. Symp. on Computer Architecture (ISCA)*, 2018, pp. 776–789.

[14] G. Pekhimenko, V. Seshadri, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "Base-delta-immediate compression: practical data compression for on-chip caches," in *Proc. ACM Int. Conf. on Parallel Architectures and Compilation Techniques (PACT)*, 2012, p. 377.

[15] M. Ekman and P. Stenstrom, "A robust main-memory compression scheme," in *ISCA*, 2005, pp. 74–85.

[16] E. Hallnor and S. Reinhardt, "A Unified Compressed Memory Hierarchy," in *HPCA*, 2005, pp. 201–212.

[17] R. B. Tremaine, P. A. Franaszek, J. T. Robinson, C. O. Schulz, T. B. Smith, M. E. Wazlowski, and P. M. Bland, "IBM Memory Expansion Technology (MXT)," *IBM Journal of Research and Development*, vol. 45, no. 2, pp. 271–285, 2001.

[18] B. Abali, H. Franke, Xiaowei Shen, D. Poff, and T. Smith, "Performance of hardware compressed main memory," in *HPCA*, 2001, pp. 73–81.

[19] M. Rhu, N. Gimelshein, J. Clemons, A. Zulfiqar, and S. W. Keckler, "vDNN: Virtualized deep neural networks for scalable, memory-efficient neural network design," in *MICRO*, 2016, pp. 1–13.

[20] Y. Kwon and M. Rhu, "Beyond the memory wall: a case for memory-centric HPC system for deep learning," in *MICRO*, 2018, pp. 148–161.

[21] M. Rhu, M. O'Connor, N. Chatterjee, J. Pool, Y. Kwon, and S. W. Keckler, "Compressing DMA Engine: Leveraging Activation Sparsity for Training Deep Neural Networks," in *Proc. IEEE Int. Symp. on High-Performance Computer Architecture (HPCA)*, 2018, pp. 78–91.

[22] S. Tokui *et al.*, "Chainer: A deep learning framework for accelerating the research cycle," in *Proc. ACM/SIGKDD Int. Conv. on Knowledge Discovery & Data Mining*. ACM, 2019, pp. 2002–2011.

[23] A. Paszke *et al.*, "Automatic differentiation in PyTorch," in *NeurIPS Autodiff Workshop*, 2017.

[24] S. Ioffe and C. Szegedy, "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift," in *ICML*, 2015.

[25] S. Zagoruyko and N. Komodakis, "Wide Residual Networks," *arXiv:1605.07146 [cs]*, 2016.

[26] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications," *arXiv:1704.04861v1 [cs.CV]*, 2017.

[27] K. Simonyan and A. Zisserman, "Very Deep Convolutional Networks for Large-Scale Image Recognition," *arXiv:1409.1556 [cs]*, 2014.

[28] Y. Chen *et al.*, "DaDianNao: A Machine-Learning Supercomputer," in *MICRO*, 2014, pp. 609–622.

[29] Y.-H. Chen, T.-J. Yang, J. Emer, and V. Sze, "Eyeriss v2: A Flexible Accelerator for Emerging Deep Neural Networks on Mobile Devices," *arXiv:1807.07928 [cs]*, 2018.

[30] A. Delmas Lascorz *et al.*, "Bit-Tactical: A Software/Hardware Approach to Exploiting Value and Bit Sparsity in Neural Networks," in *Proc. ACM Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019, pp. 749–763.

[31] J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger, and A. Moshovos, "Cnvlutin: Ineffectual-Neuron-Free Deep Neural Network Computing," in *ISCA*, 2016, pp. 1–13.

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

[33] ——, "BinaryConnect: Training Deep Neural Networks with binary weights during propagations," in *NeurIPS*, 2015, pp. 3123–3131.

[34] S. Wu, G. Li, F. Chen, and L. Shi, "Training and Inference with Integers in Deep Neural Networks," in *ICLR*, 2018.

[35] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, "Binarized Neural Networks," in *NeurIPS*, 2016, pp. 4107–4115.

[36] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, "XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks," in *Proc. European Conf. on Computer Vision*, 2016, pp. 525–542.

[37] F. Li, B. Zhang, and B. Liu, "Ternary Weight Networks," *arXiv:1605.04711 [cs]*, 2016.

[38] D. Williamson, "Dynamically scaled fixed point arithmetic," in *Proc. IEEE Pacific Rim Conf. on Communications, Computers, and Signal Processing*, 1991, pp. 315–318.

[39] A. Robinson and C. Cherry, "Results of a prototype television bandwidth compression scheme," *Proc. IEEE*, vol. 55, no. 3, pp. 356–364, 1967.

[40] Y. Zhang, J. Yang, and R. Gupta, "Frequent value locality and value-centric data cache design," in *ASPLOS*, 2000, pp. 150–159.

[41] G. K. Wallace, "The JPEG still picture compression standard," *IEEE Transactions on Consumer Electronics*, vol. 38, no. 1, 1992.

[42] M. Brenon and C. Deltheil, "A lightweight and portable JPEG encoder written in C.: Moodstocks/jpec," 2018, original-date: 2012-01-06. [Online]. Available: https://github.com/Moodstocks/jpec

[43] M. Rabbani, "JPEG2000: Image Compression Fundamentals, Standards and Practice," *Journal of Electronic Imaging*, vol. 11, no. 2, 2002.

[44] J. Aczl and Z. Darczy, *On Measures of Information and Their Characterizations*. Academic Press, 1975.

[45] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt, "Analyzing CUDA workloads using a detailed GPU simulator," in *Proc. IEEE Int. Symp. on Performance Analysis of Systems and Software (ISPASS)*, 2009, pp. 163–174.

[46] J. Lew *et al.*, "Analyzing Machine Learning Workloads Using a Detailed GPU Simulator," in *ISPASS*, 2019, pp. 151–152.

[47] M. Khairy, J. Akshay, T. Aamodt, and T. G. Rogers, "Exploring Modern GPU Memory System Design Challenges through Accurate Modeling," *arXiv:1810.07269 [cs]*, 2018.

[48] Z. Jia, M. Maggioni, B. Staiger, and D. P. Scarpazza, "Dissecting the NVIDIA Volta GPU Architecture via Microbenchmarking," *arXiv:1804.06826 [cs]*, 2018.

[49] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, and D. Kalenichenko, "Quantization and training of neural networks for efficient integer-arithmetic-only inference," in *CVPR*, 2018.

[50] H. Kim, H. Nam, W. Jung, and J. Lee, "Performance analysis of CNN frameworks for GPUs," in *ISPASS*, 2017, pp. 55–64.

[51] M. Abadi *et al.*, "Tensorflow: A system for large-scale machine learning," in *Proc. USENIX Symp. on Operating Systems Design and Implementation (OSDI)*, 2016, pp. 265–283.

[52] C. Loeffler, A. Ligtenberg, and G. Moschytz, "Practical fast 1-D DCT algorithms with 11 multiplications," in *Proc. Int. Conf. on Acoustics, Speech, and Signal Processing*, 1989, pp. 988–991 vol.2, iSSN: 1520-6149.

[53] D. Lundgren, "JPEG Encoder Verilog," 2009. [Online]. Available: https://opencores.org/projects/jpegencode

[54] H. Ishihara, "JPEG Decoder," 2006. [Online]. Available: https://opencores.org/projects/djpeg

[55] I. J. Goodfellow, J. Shlens, and C. Szegedy, "Explaining and Harnessing Adversarial Examples," *arXiv:1412.6572 [cs, stat]*, 2015.

[56] H. Li, Z. Xu, G. Taylor, C. Studer, and T. Goldstein, "Visualizing the Loss Landscape of Neural Nets," in *NeurIPS*, 2018, pp. 6389–6399.

[57] E. Hoffer, I. Hubara, and D. Soudry, "Train longer, generalize better: closing the generalization gap in large batch training of neural networks," *arXiv:1705.08741 [cs, stat]*, 2017.

[58] A. Krizhevsky, "Learning multiple layers of features from tiny images," Master's thesis, Univ. of Toronto, 2009.

[59] E. Agustsson and R. Timofte, "Ntire 2017 challenge on single image super-resolution: Dataset and study," in *CVPR Workshops*, 2017.

[60] S. Saito, "chainer-cifar10: Various CNN models including for CIFAR10 with Chainer," 2018, original-date: 2015-06-09T14:39:43Z. [Online]. Available: https://github.com/mitmul/chainer-cifar10

[61] J. Kim, J. Kwon Lee, and K. Mu Lee, "Accurate Image Super-Resolution Using Very Deep Convolutional Networks," in *CVPR*, 2016, pp. 1646–1654.

[62] J. E. Stine *et al.*, "Freepdk: An open-source variation-aware design kit," in *Proc. IEEE Int. Conf. on Microelectronic Systems Education (MSE)*, 2007, pp. 173–174.

[63] M. Martins, J. M. Matos, R. P. Ribas, A. Reis, G. Schlinker, L. Rech, and J. Michelsen, "Open Cell Library in 15nm FreePDK Technology," in *Proc. Int. Symp. on Physical Design*, 2015, pp. 171–178.

[64] P. Mattson *et al.*, "Mlperf training benchmark," 2019.

[65] NVIDIA Corporation, "Tesla V100 Datacenter GPU: NVIDIA," 2018. [Online]. Available: https://www.nvidia.com/en-us/data-center/tesla-v100

[66] S. Han, H. Mao, and W. J. Dally, "Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding," *arXiv:1510.00149v5 [cs.CV]*, 2016.

[67] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, "SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5mb model size," *arXiv:1602.07360 [cs]*, 2016.

[68] Y. Wang, C. Xu, S. You, D. Tao, and C. Xu, "CNNpack: Packing Convolutional Neural Networks in the Frequency Domain," in *NeurIPS*, 2016, pp. 253–261.

[69] C. Ding *et al.*, "CirCNN: Accelerating and Compressing Deep Neural Networks Using Block-circulant Weight Matrices," in *MICRO*, 2017, pp. 395–408.

[70] E. Park, J. Ahn, and S. Yoo, "Weighted Entropy Based Quantization for Deep Neural Networks," in *CVPR*, 2017, pp. 7197–7205.

[71] Y. Lin, S. Han, H. Mao, Y. Wang, and W. J. Dally, "Deep Gradient Compression: Reducing the Communication Bandwidth for Distributed Training," *arXiv:1712.01887 [cs, stat]*, 2017.

[72] M. Jaderberg, W. M. Czarnecki, S. Osindero, O. Vinyals, A. Graves, D. Silver, and K. Kavukcuoglu, "Decoupled Neural Interfaces using Synthetic Gradients," *ArXiv e-prints*, 2016.