ShapeShifter: Enabling Fine-Grain Data Width Adaptation in Deep Learning

Alberto Delmás Lascorz
University of Toronto
delmasl1@ece.utoronto.ca

Sayeh Sharify
University of Toronto
sayeh@ece.utoronto.ca

Isak Edo
University of Toronto
isak.edo@mail.utoronto.ca

Dylan Malone Stuart
University of Toronto
dylan.stuart@mail.utoronto.ca

Omar Mohamed Awad
University of Toronto
omar.awad@mail.utoronto.ca

Patrick Judd*
University of Toronto
pjudd@nvidia.com

Mostafa Mahmoud
University of Toronto
mostafam@eecg.utoronto.edu

Milos Nikolic
University of Toronto
milos.nikolic@mail.utoronto.ca

Kevin Siu
University of Toronto
kcm.siu@mail.utoronto.ca

Zissis Poulos
University of Toronto
zpoulos@ece.utoronto.ca

Andreas Moshovos
University of Toronto
moshovos@ece.utoronto.ca

ABSTRACT

We show that selecting a data width for *all* values in Deep Neural Networks, quantized or not and even if that width is different per layer, amounts to worst-case design. Much shorter data widths can be used if we target the *common case* by adjusting the data type width at a much finer granularity. We propose *ShapeShifter*, where we group weights and activations and encode them using a width specific to each group and where typical group sizes vary from 16 to 256 values. The per group widths are selected statically for the weights and dynamically by hardware for the activations. We present two applications of *ShapeShifter*. In the first, that is applicable to any system, *ShapeShifter* reduces off- and on-chip storage and communication. This *ShapeShifter*-based memory compression is simple and low cost yet reduces off-chip traffic to 33% and 36% for 8-bit and 16-bit models respectively. This makes it possible to sustain higher performance for a given off-chip memory interface while also boosting energy efficiency. In the second application, we show how *ShapeShifter* can be implemented as a surgical extension over designs that exploit variable precision in time.

ACM Reference Format:
Alberto Delmás Lascorz, Sayeh Sharify, Isak Edo, Dylan Malone Stuart, Omar Mohamed Awad, Patrick Judd, Mostafa Mahmoud, Milos Nikolic, Kevin Siu, Zissis Poulos, and Andreas Moshovos. 2019. ShapeShifter: Enabling Fine-Grain Data Width Adaptation in Deep Learning. In *The 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-52), October 12–16, 2019, Columbus, OH, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3352460.3358295

*Also with NVIDIA. Work completed while at the University of Toronto.

MICRO-52, October 12–16, 2019, Columbus, OH, USA

ACM ISBN 978-1-4503-6938-1/19/10…$15.00
https://doi.org/10.1145/3352460.3358295

1 INTRODUCTION

While training Deep Learning models requires some floating point arithmetic [1, 2], the models can be trained to use a different data type during inference. For several tasks, e.g., image classification, fixed-point arithmetic has proven sufficient, with simple down-conversion to 16-bit fixed point (`int16`) arithmetic being generally applicable. Such down-conversion is a rudimentary form of *quantization*. Quantization to the shortest data width possible is especially appealing for Deep Learning workloads for two reasons: First, most of their energy expenditure is due to data transfers. Using a short data type reduces the volume of this data. Second, since the data is input to numerous multiply-accumulate (MAC) operations which exhibit great data parallelism, the shorter the data type, the more functional units we can deploy for a given on-chip area and the higher performance and energy efficiency.

Given the immediate benefits on commodity hardware and accelerators alike, quantization has naturally attracted a lot of attention. As a result, today, quantization to `int8` is usually possible for state-of-the-art image classification models, and selectively in other application domains [3–11]. Quantization to shorter data widths such as 4b or 2b [7, 12, 13], or even to 1b has been demonstrated in certain cases [14–17].

Regardless, the aforementioned quantization methods target a fixed data width either per network or per layer. We observe that data widths can be trimmed further since: 1) by design, the expected per-layer distribution of values in deep learning models, be it for weights or activations, is that most will be near zero and few will be of higher magnitude, and 2) which values will assume a high magnitude will vary with the input. Targeting a specific data width for all values, even if this is adjusted per layer [18–25], disproportionately favors the exceedingly few high magnitude values, incurring

significant overheads for the vast majority of data transfers and computations. However, the data width used at any given point of time needs to accommodate only 1) the activation values for the *specific* input at hand, and further 2) the activation and weight values that are being processed or transferred concurrently. Accordingly, we propose mechanisms to exploit the expected distribution of values in neural nets to deliver memory footprint, traffic, and computation throughput improvements in a way that is generally applicable to *any* network, regardless of whether any specific form of quantization was applied or is possible.

The key idea proposed in this work is to use *per group data width adaptation*, where a group is a set of values that are either calculated upon or transferred from/to memory together. For example, these could be 16 values that are read from off-chip memory or 32 values that are being processed together in a SIMD style execution unit. We propose to select the data width dynamically for activations and statically for weights. For example, if we are storing 16 values from an `int8` network where the maximum magnitude among them is `0x3` our goal is to use only 2b per value plus some metadata to store them in memory. For another group where the maximum magnitude happens to be `0xf` we wish to use instead 4b per value. We further wish to develop a compute unit where if processing a group where the maximum magnitude requires 8b takes C time, processing the two aforementioned groups would require instead just $\frac{C}{8} \times 2$ and $\frac{C}{8} \times 4$ time respectively. Note that block-floating point[26], flexpoint [27] and narrow floating point formats [28] do not take advantage of the expected distribution of the neural net values to adapt data width. Floating-point representations expand the value range but still represent all values using the same number of bits. Moreover, most floating-point proposals target training, and generally fixed-point is a lower cost alternative for inference.

Our first contribution is to show that adjusting the data width per group dynamically for activations and statically for weights yields much shorter effective data widths than even per layer width selection. It does so *without affecting numerical fidelity* and for that accuracy. **As our second contribution** we bring attention to a property of popular quantization methods which is that while they successfully "squeeze" broader value ranges to fit within a target data width, they also *"expand"* shorter data ranges into the target data width. This expansion is often unnecessary and obscures opportunities for per group data width reduction. We deploy a range-aware quantization method, preserving the benefits of per group data length adaptation.

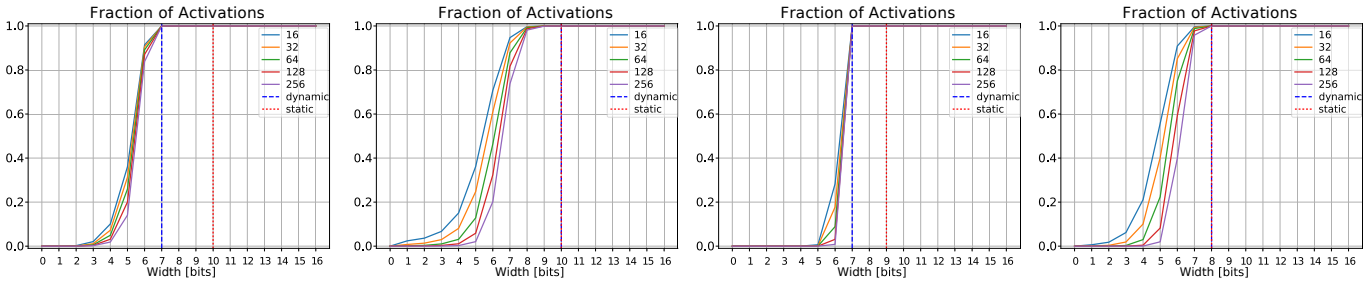
Our first novel hardware technique is a low cost *lossless* memory compression method that is plug-in compatible with most Deep Learning hardware. Our low cost *ShapeShifter* hardware compresses and decompresses groups of weights and activations off-chip reducing energy considerably and boosting the effective off-chip bandwidth. For weights, the compression is done once as a pre-processing step in software. For activations, it is performed dynamically at the output of the previous layer or at the source. Compressed data is decompressed on-the-fly when fetched from off-chip. To show that our technique is compatible with most hardware, we demonstrate benefits over a range of accelerators.

Our second novel hardware technique adapts the width of activations and weights at runtime so that execution time is proportionally reduced per group of concurrently processed values. This is compatible with designs that exploit data width variability *in time*, e.g., [29–31] as opposed to designs that do so *spatially* [25]. We present *SStripes*, a surgical extension over the *Stripes* accelerator [29]. A major advantage of our proposal is that it requires modest hardware changes yet the resulting performance, communication, storage, and energy efficiency improvements are anything but. Moreover, the area overhead is exceedingly small.

Our *ShapeShifter* approach is not a quantization method and does not affect numerical range, value nor accuracy. It simply adjusts the data container width to accommodate the values at hand at a fine, programmer transparent granularity. Quantization methods transform the values so that they fit a particular data width. For virtually all quantization methods, the width has to be large enough to accommodate *all* values within a large set of data, typically a layer or the whole network. For this reason there will be values that in practice could be represented in a narrower width and thus there is going to be opportunity for *ShapeShifter*. Naturally, the smaller the quantization width, the smaller the relative opportunity will be. But, unless it becomes possible to quantize all networks of interest to a small data width, *ShapeShifter* delivers proportional benefits for all networks. Unique among quantization methods, is outlier-aware quantization [32] which uses two data widths. Most values (97% to 99%) use the shorter width (e.g., 4b or 5b), and only the remaining few use the full data width (e.g., 8b or 16b). The two data widths are fixed, design time parameters and the network has to be retrained explicitly for those. We show that *ShapeShifter* directly benefits from this form of quantization providing additional benefits by further reducing data width for both sets of values. However, contrary to outlier-aware quantization *ShapeShifter* works with any model, quantized or not and imposes no design or runtime constraints on the data widths used, their relative frequeny or their spatial distribution.

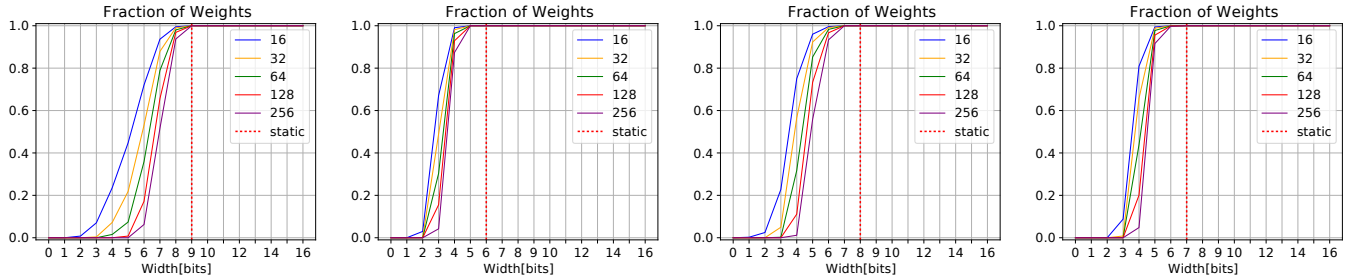
We highlight the following findings:

- *ShapeShifter* off-chip memory compression reduces traffic to 27% and 36% respectively for the 16b and the range-aware quantized 8b models studied. With Tensorflow quantization, the benefits are much lower as traffic is reduced to at most 80% for some of the models only. *ShapeShifter* compression is robust and never increases traffic.
- *ShapeShifter* compression boosts the effective memory bandwidth yielding energy savings and boosts performance, e.g., performance for BitFusion improves by 87% with DDR4-3200 memory for 16b models.
- *ShapeShifter* Stripes (*SStripes*) improves compute performance over Stripes by 1.61× for 16b models, and 2.17× or 1.49× for 8b models depending on the quantization method (Range-Aware or Tensorflow respectively). It is also faster than Bit Fusion by 3.75× and 2.3× for the 8b models.
- *ShapeShifter* compression delivers virtually all the memory traffic reduction possible on outlier-aware quantized models despite not being specialized for them. It further boosts compression rates by 24.5% on average.



(a) GoogLeNet, Conv 1 (b) GoogLeNet, 5a-1x1 (c) ResNet50, res3a_branch1 (d) ResNet50, res5a_branch1

Figure 1: Per Group vs. Per Layer Activations Width Needs: 16b Models



(a) GoogLeNet, Conv 1 (b) GoogLeNet, 5a-1x1 (c) ResNet50, res3a_branch1 (d) ResNet50, res5a_branch1

Figure 2: Per Group vs. Per Layer Weights Width Needs: 16b Models

2 DATA WIDTH NEEDS

This section demonstrates that, for a variety of 16b and 8b models: 1) the widths needed at a finer-than-a-layer granularity are considerably shorter than those needed for the whole layer, 2) only a small number of value groups require the maximum width needed for the layer as a whole, 3) the width needed varies with the input, and 4) some quantization methods may unnecessarily expand value ranges to the full range afforded by the target width.

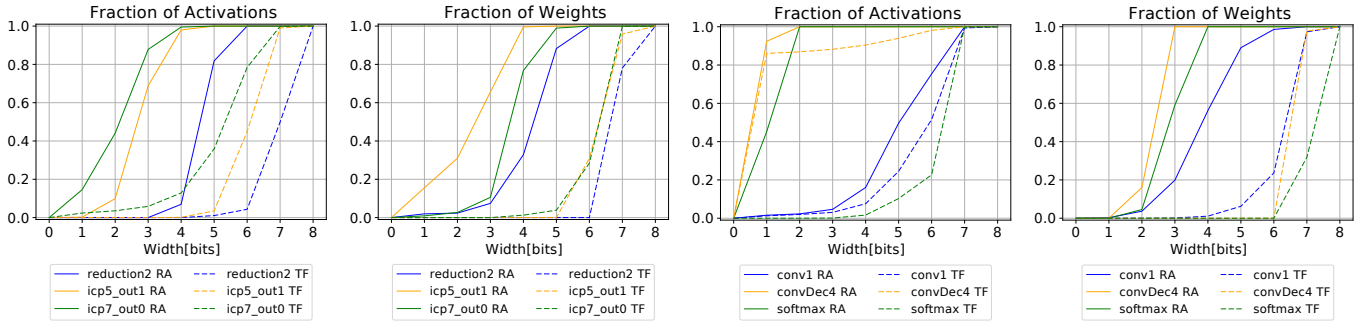
Activations: Figures 1a-1d show data width (henceforth referred to as just *width*) measurements for four convolutional layers – two from each of GoogleNet and SkimCaffe-ResNet50 [33] (a weight pruned network), both originally quantized to 16b. Similar trends were observed in other 16b models and their layers. The measurements are over 5,000 randomly selected images from the IMAGENET dataset [34]. The graphs show the cumulative distribution of the width needed per group of activations for various group sizes ranging from 16 to 256 values, where each group uses the width needed for its least favorable activation. Two vertical lines report the width when considering all activation values: 1) the profile-derived ("static") over all images, and 2) the one that can be detected dynamically ("dynamic") for one randomly selected image to illustrate that widths also vary per input.

Figure 1a reports measurements for `conv1`, the first layer of GoogleNet. The profile-determined width is 10b while for one specific image all values within the layer could be represented with just 7b, an improvement of 30% in width length. The cumulative distribution of the widths needed per group of 256 activations shows that further reduction in width is possible. For example, about 80% of these groups require 6b only, and about 15% just 5b. Smaller group sizes further reduce the *effective* width, however, the differences are modest. These results suggest that picking a width for the whole layer exacerbates the importance of a few high-magnitude activations.

The first layer usually exhibits different value behavior than the rest since it processes the direct input image to discover visual features whereas the inner layers tend to try to find correlations among features. Layer `5a-1x1` in Figure 1b shows more pronounced improvements in effective width with the smaller group sizes. Figures 1c and 1d show that the behavior persists even in a sparse network. We include this measurement to demonstrate that dynamic width variability is a phenomenon that is orthogonal to weight sparsity and thus of potential value to designs that target sparsity. Moreover, dynamic width variability is not only limited to activations.

Weights: Figure 2a reports measurements for the weights in the first layer of GoogleNet. While the profiled-determined width is 9b, dynamic precision determines that 80% of the groups can be represented using only 8b for all group sizes. For a group size of 16, 70% of the weight groups require only 6b. This is more pronounced for Figure 1b where 80% of the bits can be represented using only 4b rather than the 6b profiled. Figures 2c and 2d show that this behaviour persists in sparse networks.

8b Quantization: Figure 3a reports the distribution of widths needed for GoogleNetS quantized to 8b under two quantization methods: 1) Tensorflow (TF) and 2) Range Aware (RA). The figure shows that Tensorflow may unnecessarily expand the value range



(a) GoogLeNetS Activations (b) GoogLeNetS Weights (c) SegNet Activations (d) SegNet Weights

Figure 3: Per Group Data Width Needs for 8b models with Tensorflow (TF) and Range Aware (RA) quantization.

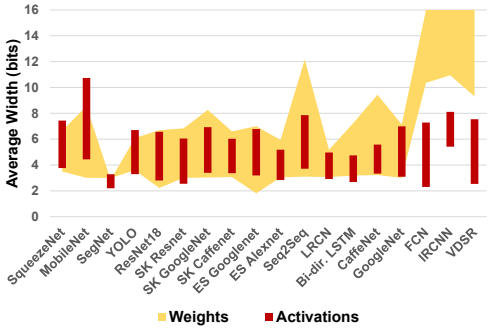
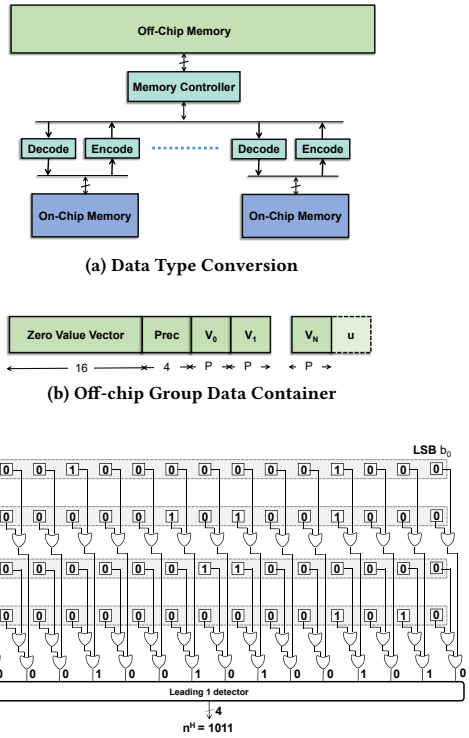


Figure 4: Average data width needed for Weights and Activations: Per Layer (top) vs. Per Value (bottom).



(a) Data Type Conversion

(b) Off-chip Group Data Container

(c) Detecting per group widths for activations.

Figure 5: Using per group widths to reduce off-chip memory bandwidth and storage.

to 8b even if a layer does not need to, whereas Range-Aware quantization does not. The figure shows three of the longer running layers which are representative of the range of behaviors seen over all layers (best, average, and worst). Under range-aware quantization, most values require considerably less than 8b. In `reduction2` and `icp5_out1` more than 80% of the activations need just 3b or less, a reduction of at least 87.5% over 8b. Even in `icp7_out0`, the most demanding layer, more than 90% of the activations need only 6b - a 25% reduction over 8b. Under Tensor flow quantization, the same model exhibits much higher data width needs. Even for `reduction2`, 96% of the values now need 6b. Figure 3b shows unnecessary value expansion by the Tensorflow quantization also for the weights. In the case of `icp5_out1`, Tensorflow needs the whole 8b to represent all the groups, in contrast to Range Aware quantization requires only 4b. Figures 3c and 3d show similar behavior for SegNet.

Other Models: Figure 4 demonstrates that width requirements vary considerably between the layer and individual values for a broad range of neural network models. It compares the average effective width with per layer profiling or with per value detection for weights and activations for a variety of models. The top point of the surface for weights, and the bar for activations, corresponds to the per layer and the per value widths respectively. These measurements are for 1,000 randomly selected images from IMAGENET [34] for the image classification models [33, 35–39] and for 100 images from CamVid [40] for SegNet [41](segmentation), 100 and 10 inputs from Pascal VOC [42] respectively for YOLO V2 [43] (detection) and FCN8 [44] (segmentation), 10 images from CBSD68 [45–47], 10 inputs for IRCNN [48] (denoising) and VDSR [49] (super-resolution), 500 inputs from WMT14 for Seq2Seq [50] (translation), 500 inputs from COCO [51] for LRCN [52] (captioning), and 500 inputs from Flickr8k [53] for Bi-Directional LSTM [54] (captioning). The

measurements are across the whole network. It also reports the reduction in work (left Y-axis) for weights and activations when per-value width is used. The results illustrate that selecting a width per layer grossly overestimates that needed for the common case.

Finally, Table 1 reports the *effective* data width achieved per layer. Due to space limitations we report results for the 16b models with a group size of 16 values along the channel dimension. The effective width values are fractional as they are averaged over all groups within the layer and weighted accordingly to their runtime use.

3 REDUCING OFF-CHIP STORAGE AND COMMUNICATION

The bulk of energy in DNNs is expended by off- and on-chip memory accesses. Further, when on-chip storage is limited, off-chip bandwidth can easily be the bottleneck. Fortunately, the variable width needs of activations and weights can be used to reduce the amount of storage and bandwidth needed on- and off-chip. Here we limit attention to the off-chip compression scheme.

Off-chip Memory Data Container: We encode weights and activations in groups of N values. We find that $N = 16$ offers a good balance between compression rate and metadata overhead. Figure 5b shows the memory container. For each group, we determine the width in bits, p_i, that the group needs either statically (for weights) or dynamically (using the hardware unit of Figure 5c) at the output of the previous layer or at the input source for the first layer (for the activations). We then store that as a prefix using $log(P)$ bits, where P represents the maximum data width, followed by the 16 values each stored using p_i bits. In addition, to avoid storing zero values altogether, we use a 16b "is zero" vector with one bit per original value and store only the non-zero values off-chip. In total, this scheme requires 4 + 16 bits of metadata per group of sixteen 16b values. Uncompressed these sixteen values occupy 256b for 16b values (or 128b for 8b values) so the metadata overhead is far less than the benefits obtained from compressing the values for typical cases. Figure 6 shows an example of how two groups each of eight 8b values are represented and decoded with our approach.

Detecting the Per Group widths for Activations: Figure 5c shows how the hardware adjusts the width at runtime for an example group of four 16b activations A_0 through A_3. This is needed when it is necessary to store the values off-chip at the output of each layer. It trims the unnecessary prefix bits by detecting the most significant bit position needed to represent all values within the group. The example activations can all be represented using just 12 bits as the highest bit position a 1 appears, n_H, is position 11. The hardware calculates 16 signals, one per bit position, each being the OR of the corresponding bit values across all four activations. The implementation uses OR trees to generate these signals. A "leading 1" detector identifies the most significant bit that has a 1, and reports its position in 4 bits. The detector can be extended to handle negative values by converting them first to a sign-magnitude representation, and placing the sign at the rightmost (least significant) place. This is useful for weights and for networks that use activation functions that attenuate negative values [55, 56].

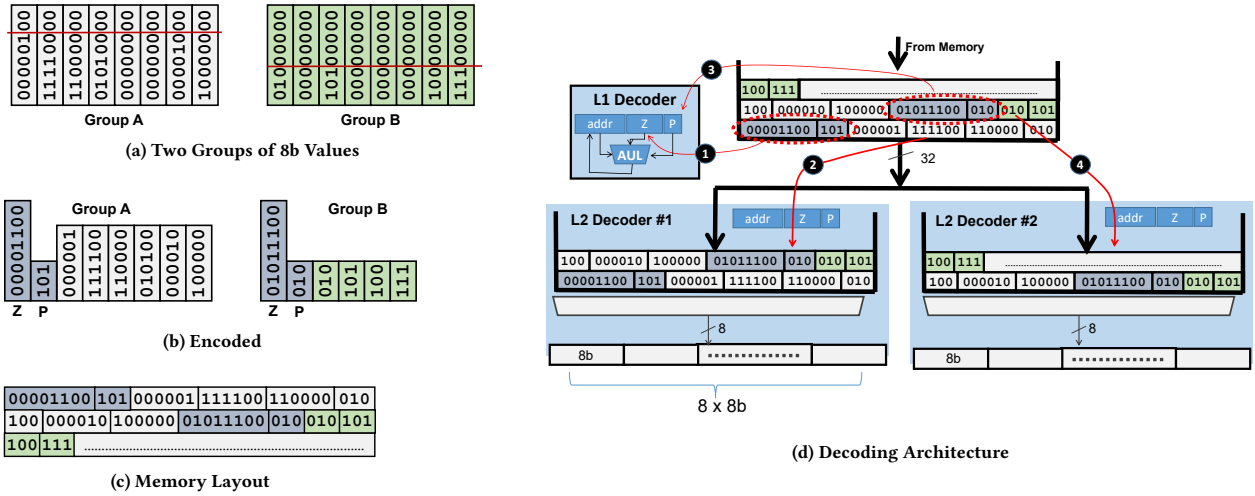
Memory Layout and Access Strategy: Memory compression and bandwidth amplification methods for general purpose programs have to support random accesses at cache block aligned addresses, e.g., [57] . For this reason they need to be able to quickly determine where each compressed block is stored, how long it is, and to handle cases where the compressed block needs more space than when uncompressed. By taking advantage of the unique requirements of neural networks, Shapeshifter compression has to meet considerably looser requirements. This is easier to understand in the case where we can size the on-chip memory buffers so that each weight and activation needs to be accessed from off-chip *only once* per layer [58]. This is practical for all networks studied. In this case, for each layer the accelerator reads from off-chip two input arrays (weights and activations) and writes a single output array (output activations). The off-chip accesses to each array can be sequential. As a result Shapeshifter needs to support: 1) direct accesses to the beginning of three large containers (whole activation or weight arrays), and 2) sequential accesses within these containers. Within each container data can be compressed as desired without concern about how much space it occupies and where it is stored: the incoming stream will be decoded sequentially.

When the on-chip memory is not sufficiently large, we have to access some activations and/or weights multiple times. To reduce off-chip traffic, an appropriate dataflow will be used and naturally such a dataflow will access data in large blocks from off-chip in order to minimize energy. The starting addresses (access handles) of these blocks/containers can be easily identified statically and accesses within each container can remain sequential. Generally, for any given dataflow it will be possible to identify access handles either statically for the weights or dynamically for the activations and pre-record those in a small separate table.

Decompression/Compression: For clarity, let us first describe decompression as a sequential process, assuming a group size of 8 and and a maximum datawidth of 8b as per the example of Figure 5a The accelerator has its own memory request engine which is issuing requests to the memory controller. The access engine maintains a base register plus length for each input array (normally one for activations and one for weights) plus a sequential address generator. The number of arrays, their length, and how the rate of requests should be interleaved is metadata that comes with the layer and is not different than what would be used for an uncompressed model.

Starting from the beginning of an activation or weight array, the decompressor reads the first 8+3=(Z, P) bits containing the metadata for the first group of 8 values. Walking through the Z one bit a time, the decompressor either outputs a zero or reads the next P bits from memory expanding to the data width used on-chip. Upon finishing with the current group, the decoder has arrived at the header for the next group and the process repeats.

Rather than using a single, sequential decompressor we use multiple ones organized in a two level hierarchy as shown in Figure 5a and detailed in Figure 6d. A single, first level decompressor (L1D) identifies where groups start handing off the per group value decompression to the second level decompressors (L2Ds), one per on-chip memory bank. Specifically, upon inspecting a group's (Z, P) header ❶, the L1D immediately determines which lines in its buffer contain the payload for the current group. It hands-off decompression to an L2D by 1) communicating $(addr, Z, P)$ where $addr$ is the starting address (line, and bit offset) for the group, and 2) copying the relevant lines from its buffer to the L2D's private buffer ❷. At the same time, the L1D can then determine where the next groups



(a) Two Groups of 8b Values

(b) Encoded

(c) Memory Layout

(d) Decoding Architecture

Figure 6: An example how *ShapeShifter* encoding/decoding works. (a): Two groups of eight 8b values. In group A only 6 bits are needed and in group B 3b are sufficient. (b): The encoded groups. Two metadata fields are introduced. Z is a bit vector that identifies zero values. These zero values are not stored in memory. P represents the number of bits used to store the remaining non-zero values. (c): The groups are stored in memory back-to-back in the order we expect them to be read. (d): Decoding using two decoders. The encoded groups are fetched from off-chip memory in order and are placed first into a per memory interface buffer. The example assumes a 32b wide external memory interface for clarity. The on-chip controller inspects the header of the first group and copies the rows containing the first group to the internal buffer of the first decoder. The rows containing the next group are then copied to the second decoder along with an index indicating where the group starts. Each decoder then expands each assigned group one value at a time and places the result into its output register. In our example, the output register contains eight 8b values. *ShapeShifter* is completely transparent to the on-chip execution engine.

Table 1: Average Per Layer widths with *ShapeShifter*.

Network	Effective Activation Width Per Layer	Reduction
AlexNet	6.52-4.7-3.48-3.23-2.68-2.19-2.59-2.35	41.09%
GoogLeNet	7.42-5.14-5.05-4.01-4.01-3.03-4.01-3.34-4.47 -4.26-4.26-3.86-3.34-5.14-3.99-3.96-3.96-4.2 -3.96-2.51-4.78-2.27-2.99-3.4-2.99-2.7-3.39-5.24 -3.36-3.41-3.36-2.66-4.18-4.08-4.08-3.01-3.18 -1.67-3.14-2.96-2.96-3.04-2.96-1.87-3.34-3.99 -2.3-2.11-3.1-2.5-4-3.85-2.31-1.79-1.65-1.33-2.29	44.34%
VGG_M	6.37-3.67-2.51-2.25-2.63-1.94-2.39-2.32	50.23%
VGG_S	5.39-3.71-3.67-2.25-2.44-1.52-2.43-3.06	46.35%
ResNet50	6.44-6.21-5.21-3.81-4.27-3.78-3.34-3.01-4.03 -3.08-3.78-4.09-3.14-3.35-3.45-4.02-2.86-3.15 -4.06-2.95-2.65-3.06-2.18-2.79-3.32-3.32-2.36 -3.27-3.16-1.97-1.98-3.06-2.43-1.96-3.01-2.24 -1.79-2.94-1.54-2.33-3.83-1.65-2.45-4.01-3.05 -1.73-2.27-2.55-1.93-1.83-2.36-1.74-1.65-3.26	53.46%
Yolo	4.99-6.03-5.29-5.19-4.19-6.36-4.3-5.18-2.66 -4.32-4.17-5.29-4.16-3.35-4.3-4.87-4.29-4.87 -3.98-4.85-3.09-4.29	33.52%
MobileNet	6.68-7.01-8.36-5.41-7.25-7.24-8.02-6.05-7.09 -5.94-7.71-4.77-7.84-6.44-7.3-7.12-9.5-6.15-8.54 -5.23-8.55-6.14-9.5-5.06-8.74-4.41-9.05-7.97	32.08%

Network	Effective Weight width Per Layer	Reduction
AlexNet	4.16-4.69-3.49-4.5-4.6-3.55-3.2-3.73	45.58%
GoogLeNet	5.58-6.86-6.1-4.91-5.68-4.75-3.89-4.18-5.12-5.28 -4.39-4.44-4.61-4.48-4.32-4.01-5.04-4.58-3.03 -3.88-5.01-4.57-3.68-4.95-2.87-4.31-4.82-4.8 -4.95-2.97-4.34-4.66-4.78-4.01-4.96-3.83-4.2 -4.76-3.36-4.27-4.15-3.68-4.67-4.56-3.31-3.33-3.59 -2.69-3.99-3.65-4.05-4.52-2.63-3.61-1.91-3.29-4.11	33.37%
VGG_M	4.57-3.91-4.31-3.99-3.98-3.79-2-3.17	35%
VGG_S	4.63-3.64-5.28-3.94-3.93-3.12-2.94-3.61	30.92%
ResNet50	5.6-4.9-6.53-3.97-4.43-3.62-3.37-5.24-4.55 -4.35-3.27-4.04-3.42-3.85-4.11-3.11-3.83-2.96 -2.07-3.5-3.39-4.39-3.93-3.92-3.68-2.99-3.41 -3.82-3.38-3.26-3.62-3.57-3.33-4.53-3.57-3.33 -3.49-3.75-3.3-3.6-3.83-3.31-3.63-4.11-3.66 -4.03-3.44-4.22-3.93-3.24-4.49-4.8-4.17-4.27	38.45%
Yolo	8-6.97-7-7.8-6.71-5.97-5.98-4.98-6.7-5.83 -5.74-6.81-6.7-3.99-5.98-4.98-4.98-4.98-4.79 -6.7-4.79-4.89	3.8%
MobileNet	3.88-3.3-4.91-2.11-3.96-2.76-3.68-1.95-3.39-2.53 -3.17-1.87-2.92-2.39-3.54-1.64-2.77-2.06-2.78 -2.06-2.84-1.66-2.84-2.77-3.43-2.11-3.05-1.68	68.68%

starts; the address update logic (AUL) block updates the fetch address for the level 1 decoder to point to the beginning of the next compressed group in memory (+= ones(Z) × P), inspects the new (Z, P) header ❸ and sends the corresponding set of lines to another L2D ❹.

Upon receiving a payload header from the L1D, an L2D expands the values to 8b and does so serially, one value at a time. The use of narrow read ports for the L1D and L2D buffer avoids the use of wide crossbars or shuffling networks.

Finally, at the output of each layer, the output activations are assembled in groups in each bank, and are encoded accordingly using the width detected by the hardware of Figure 5c. The memory controller writes the resulting data containers as they become available.

4 REDUCING EXECUTION TIME

Here we discuss how *ShapeShifter* can be used to reduce execution time with designs that exploit data width variability *in time*. As representative of these designs, this section explains how *ShapeShifter* can benefit Stripes [29]. Figure 7a shows an example illustrating what the goal is: with *ShapeShifter*, Stripes (*SStripes*) will adjust the number of cycles it needs to process a group of activations (8b in the example) to accommodate the value with the highest magnitude within the group. This is different than the original Stripes where each group was processed in the same number of cycles; a profile-derived precision dictated how many cycles to use.

Lets us first review how Stripes exploits per layer precisions to improve performance. Figure 7b shows a Stripes *serial inner-product* unit (SIP). The SIP multiplies and accumulates 16 (activation, weight) pairs (16b values). The SIP processes the activations one bit at time and thus its execution time varies according to the data width chosen. Any data width up to 16b is supported. For example, if the layer precision was determined to be 12 (light grey area), then the SIP will take 12 cycles per group of 16 activations.

By introducing a precision detection unit prior to feeding the activations to the SIP, it becomes possible to adjust the number of cycles needed on a per group basis. For the specific group of values, it will be possible to process the values in 8 cycles despite the precision chosen for the layer being 12.

Figure 7c shows the overall Stripes architecture identifying the *ShapeShifter* extensions. Stripes comprises several tiles. Each tile processes 16 filters and 16 weights per filter (design parameters). A local to the tile weight memory (WM) provides the weights. Each tile contains 256 SIPs organized into 16 rows of 16 columns. The SIPs along the same row process different windows of the same output channel and reuse the same set of weights. A local activation memory scratchpad (ASpad) provides the activations via 256 wires, one per activation. The SIPs along the same column process the same window across different output channels (filters). This way weights and activations are reused spatially. Further reuse is possible by tiling the computation. Partial sums are accumulated within the SIPs and eventually shipped out to per tile partial sum memories (PM). An activation memory (AM) broadcasts activations to the tiles. A dispatcher per activation memory bank takes care of transposing the values and communicating them bit-serially to the tiles. The AM can be centralized or distributed along the tiles. The dark boxes in Figure 7c identify the *ShapeShifter* extensions. A per dispatcher width detection unit (Figure 5c) inspects the activation group and communicates the data width needed via an "end of group" (EOG) signal (not shown). Processing proceeds as in the original Stripes starting from the least significant bit (which is always the same for simplicity) but may terminate early according to the EOG signal.

The second, optional extension to Stripes is the "Composer" block. It contains a $2 \times 36b$ adder every two rows. This allows the results of two SIPs adjacent along the same column to be added prior to writing them to the PM. It is to be used with SIPs that process 8b weights instead of the 16b weight SIPs originally used in Stripes — the SIPs still support up 16b activations and 36b accumulators. SIPs configured to process 8b weights are 1.8× smaller than SIPs processing 16b weights. For those layers where profiling determines

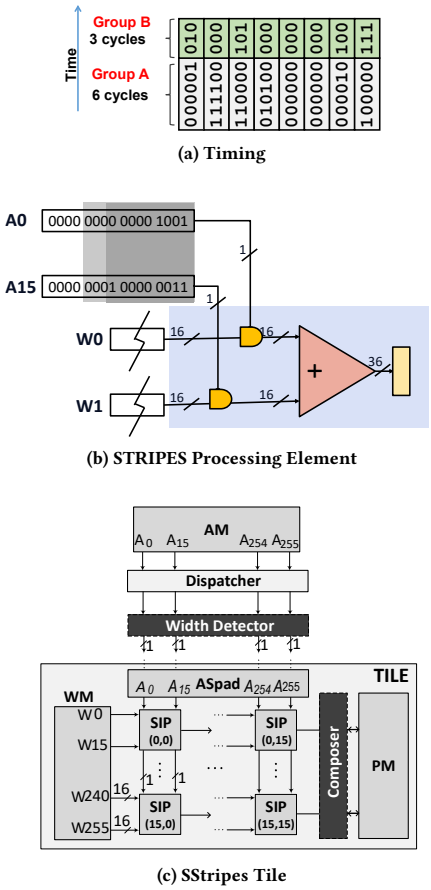


Figure 7: *SStripes*: Extending Stripes with *ShapeShifter*

that more than 8b are needed for the weights, we use two adjust along the column SIPs to process the upper and the lower 8 bits of the weights respectively. By adding the partial sums accumulated in the two SIPs we can then correctly calculate the final output activation for weights that need more than 8b in total. This is done, after processing all groups and as the two values are brought out of the SIP array prior to writing them to the PM.

Stripes exploits precisions in the time domain only. Bit Fusion instead exploits precisions spatially up 8b, and then in time for 16b values. Bit Fusion is thus a *spatial-first* design and as presented cannot adapt to precisions at a fine granularity. The composer block allows *SStripes* to exploit precisions up to 8b in time, and then up to 16b spatially. Accordingly, the resulting design is *time-first* and thus can adapt to per group precisions. Adapting a spatial-first design to adjust precisions at a fine-granularity is left for future work.

Simplicity and low cost are major advantages of our proposal. The area overhead of per group width adaptation is negligible, at below 2% compared to the tile. *SStripes* does not affect accuracy, and produces the same numerical result as Stripes.

Table 2: Neural Networks Studied. ⋆Pruned.

Model	Dataset	Application
int16		
Alexnet [59]	ImageNet [34]	Classification
Alexnet-S⋆ [37]	ImageNet [34]	Classification
Alexnet-S2⋆ [33]	ImageNet [34]	Classification
Googlenet-S⋆ [37]	ImageNet [34]	Classification
Googlenet-S2⋆ [33]	ImageNet [34]	Classification
VGG_M [60]	ImageNet [34]	Classification
VGG_S [60]	ImageNet [34]	Classification
ResNet50 [61]	ImageNet [34]	Classification
Resnet50-S⋆ [33]	ImageNet [34]	Classification
Yolo [43]	ImageNet [34]	Real-Time Object Detection
MobileNet [39]	ImageNet [34]	Classification
int8: Tensorflow Quantized		
Alexnet-S⋆ [37]	ImageNet [34]	Classification
Googlenet-S⋆ [37]	ImageNet [34]	Classification
Resnet50-S⋆ [33]	ImageNet [34]	Classification
MobileNet [39]	ImageNet12 [34]	Classification
int8: Range-Aware Quantized		
Alexnet-S⋆ [37]	ImageNet [34]	Classification
Googlenet-S⋆ [37]	ImageNet [34]	Classification
BiLSTM [54]	Flickr8k [53]	Captioning
SegNet [41]	CamVid [40]	Segmentation
int16/int4 or int16/int5: Outlier-Aware Quantized		
ResNet50 [61]	ImageNet12 [34]	Classification
MobileNet-v2 [62]	ImageNet12 [34]	Classification

5 EVALUATION

All accelerators were modeled using the same methodology for consistency. A custom cycle-accurate simulator models execution time. To estimate power and area, all designs were synthesized with the Synopsys Design Compiler [63] for a TSMC 65nm library (unfortunately, due to licensing constraints, presently 65nm is the best technology that we have access to) and laid out with Cadence Innovus. Circuit activity was captured with ModelSim and fed into Innovus for power estimation. All designs operate at 1GHz and all results reported are post-layout measurements. However, for Bit Fusion we use the power and area reported for 45nm technology [25]. The SRAM activation buffers, and the Activation and Weight memories were modeled using CACTI [64]. Three design corner libraries were considered prior to layout. The typical case library was chosen for layout since bit-serial designs are affected less by the worst-case design corner. The on-chip memories were sized so that for *most* layers it is possible to read each value from off-chip memory at most *once* per layer using the method of Siu et al., [58]. Since off-chip accesses dominate energy consumption appropriately sizing the on-chip memory is a primary design consideration [65]. We use 4MB for activations and 4MB for weights for the 8b models and double that for the 16b models. This configuration allows us to hold a complete row of input activation windows on-chip at any given point of time for most layers. Measurements are reported over all layers of each network. Table 2 details the neural networks studied. The outlier-aware models are used in Section 5.4 to demonstrate that *ShapeShifter* compression naturally supports such models delivering the memory traffic reduction that is expected but without specializing to this form of quantization.

5.1 ShapeShifter Memory Compression

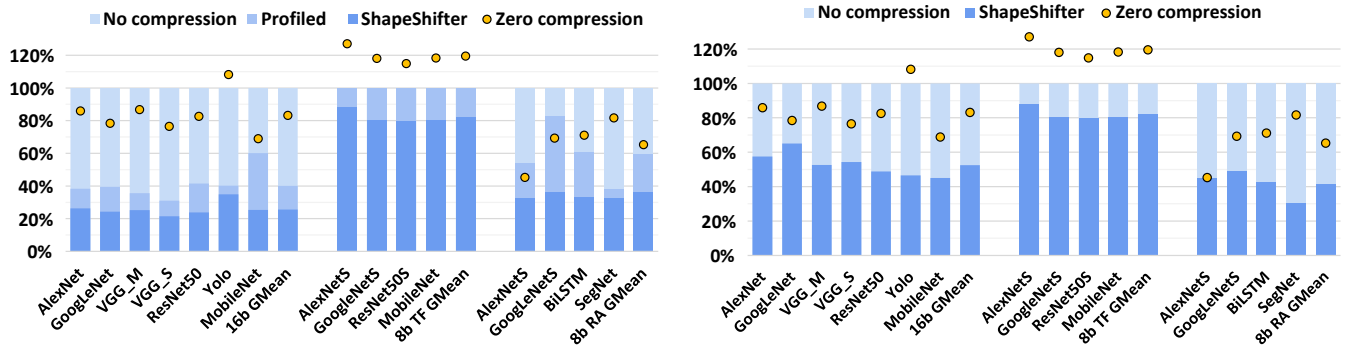
Compression Rate: Figure 8a reports relative off-chip traffic for four schemes: a) baseline, no compression ("Base"), b) compressing using per layer profile-derived widths ("Profile") [22], c) *ShapeShifter* compression ("ShapeShifter"), and d) runlength zero compression ("Zero compression") as used in Eyeriss and SCNN. The compression group for Profile and ShapeShifter is 16 values adjacent along the depth dimension. For the 16b models, *ShapeShifter* compression reduces traffic to less than 30% on average. The Tensorflow Quantized 8b networks see less benefit from our technique because they quantize in such a way that dynamic precision reduction cannot be applied effectively. However, the Range-Aware Quantized 8-bit networks see a 30-50% reduction in memory traffic. *ShapeShifter* compression also achieves better traffic reduction than zero compression.

Figure 8b reports relative off-chip traffic for non-profiled networks. While profiling is feasible it is not always possible, e.g., when the test data set is not available. *ShapeShifter* memory compression achieves a traffic reduction to less than 65% for the non-profiled 16-bit networks, while for the non-profiled 8-bit networks the traffic is reduced to 30%-50%. The Tensorflow Quantized 8-bit networks require more traffic than the dynamic precision compressed Range-Aware Quantized 8-bit networks without profiled precisions. For the sparse Range-Aware Quantized 8-bit network AlexNetS the off-chip traffic reduction matches that of zero compression and outpeforms it for all other networks.

We demonstrate how by reducing off-chip memory traffic *ShapeShifter* compression benefits DaDianNao*, Bit Fusion, and SCNN improving overall performance and energy efficiency. *ShapeShifter* compression resulted in energy savings for all off-chip memories regardless of their speed rating. To demonstrate that it can also considerably improve performance we report speedups with a lower-end DDR4-2133 off-chip memory, a higher-end DDR4-3200 off-chip memory, and a halfway between those, DDR4-2400 off-chip memory.

5.1.1 DaDianNao.* Figure 9a shows the performance of a DaDianNao variant, DaDianNao* using the three different off-chip compression schemes. *ShapeShifter* compression speedups execution by 2.4× and 2.01× for the 16b and 8b Range-Aware quantized models respectively compared to no compression with DDR4-2133. As expected, the average speedup is a modest 10% for the Tensorflow quantized models. Speedups are more pronounced for the older networks (AlexNetS and the VGG variants) as these have large fully-connected layers that are memory-bound. These layers could benefit much more from methods such as Deep Compression [66]. Even with higher bandwidth memory nodes *ShapeShifter* delivers benefits: *ShapeShifter* with DDR4-3200 achieves a 2.15× and 1.85× speedup over no compression for 16b and 8b Range-Aware quantized models respectively.

As Figure 9b shows, *ShapeShifter* greatly reduces energy compared to no compression or profiled compression. Not only does it naturally reduce the energy from memory accesses due to compression, but also reduces memory stalls saving on energy expended by idle computation units. We observe energy savings up to 78% compared to DaDianNao* without compression. SegNet is mainly



(a) Reduction for Profiled Networks

(b) Reduction for Non-Profiled Networks

Figure 8: Offchip traffic reduction with compression schemes. The leftmost networks are 16b, the middle networks are 8b Tensorflow Quantized and the rightmost networks are 8b Region-Aware Quantized.

compute-bound; therefore, memory compression offers little benefit.

5.1.2 Bit Fusion. BitFusion [25] targets precision profiled networks. It is composed of a systolic array of bit-level processing elements that can be grouped to match layer precision. This allows BitFusion to natively support per layer precisions of 8, 4, and 2 bits for both weights and activations. It also supports 16b models by decomposing them into 8b multiplications which it performs sequentially in time. Figures 9c and 9d report respectively performance and energy efficiency for BitFusion with each of the three off-chip compression schemes and with different off-chip memory technology nodes. The results demonstrate that BitFusion benefits from *ShapeShifter* compression.

5.1.3 SCNN. Figure 10 shows the performance of SCNN with *ShapeShifter* compression versus the run length encoding compression scheme used by SCNN for a set of 16b pruned neural networks – SCNN targets pruned models. On average, SCNN with *ShapeShifter* performs 9% faster and specifically for the newer ResNet50 network, it performs 29% better. The Figure also reports energy efficiency; SCNN with *ShapeShifter* is 9% more energy efficient than SCNN with the baseline compression.

5.1.4 Layer Fusion. We evaluated the benefits of *ShapeShifter* compression in combination with layer fusion [67]. As shown in Figure 11, the combination of fusion and *ShapeShifter* compression provides considerable reductions in external memory bandwidth as opposed to layer fusion alone.

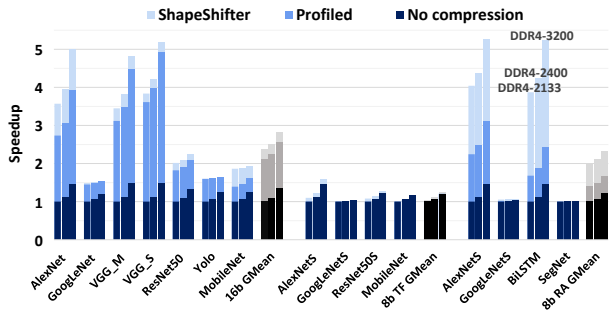
5.2 ShapeShifter Stripes

This section compares *SStripes* to *Stripes* and Bit Fusion under an iso-area constraint. To compare with Stripes, both Stripes and *SStripes* are configured to use the same total on-chip area. Stripes has 16 tiles each containing 256 serial processing units whose worst-case peak compute bandwidth is 4K multiplications per cycle. Stripes uses per layer profile-based compression [22] as originally proposed [29]. The on-chip activation and weight memories are 4MB each modeled as SRAMs for the 8b models and double that for the 16b models. These are practical and appropriate memory sizes for server class designs. By comparison the TPU1 chip has more than 20MB of on-chip storage [68]. Both designs use a dual channel DDR4-3200 memory interface which proves sufficient for delivering the necessary bandwidth for keeping the units busy. *SStripes* is configured with 8b weight/16b activation SIPs as per Section 4 plus a composer column per tile. Each *SStripes* tile contains 16x28 SIPs as they are smaller compared to the original Stripes SIPs.

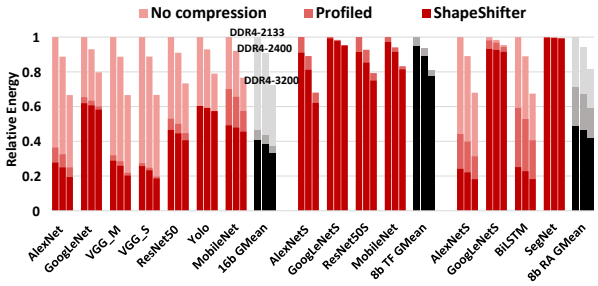
Figure 12 reports the speedup of *SStripes* over Stripes and Figure 13 reports the compute-memory breakdown for *SStripes*. All measurements are with a dual channel DDR4-3200 memory. For the 16b models, *SStripes* boosts performance by 61% on average. For the more recent ResNet50, the speedup is 88%. Older models such as AlexNet and some of the VGG models are still memory-bound, so most of the benefits come from memory compression on their relatively large fully-connected layers. MobileNet 16b accelerates the most at 2.35× since it is compute bound. For the Tensorflow quantized 8b models, benefits are lower at 49% on average. This is expected as this method forgoes opportunities to use precisions less than 8b.

The execution time breakdown shows that networks that are mostly convolutional such as SegNet, or GoogLeNet are compute-bound. Other networks such BiLSTM or those that have relatively large fully-connected layers have higher memory bandwidth requirements. While 16b MobileNet benefits from a larger reduction in memory traffic with *ShapeShifter*, the Tensorflow quantized MobileNet 8b waits for off-chip data 40% of the time. The benefits are higher at 2.17× on average for the range-aware quantized 8b models. BiLSTM benefits the most. This is due to the *ShapeShifter* memory compression working particularly well for the fully-connected and LSTM layers which are memory-bound. For SegNet which is predominantly compute-bound, *SStripes* is 93% faster where most of the benefits are from per group compute adaptation.

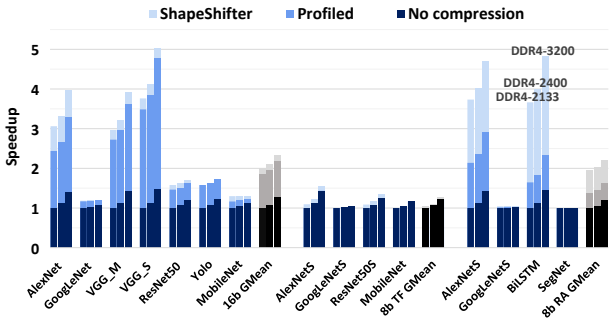
Figure 12 also reports the energy efficiency of *SStripes* over Stripes. Benefits generally follow the performance trends. Specifically, efficiency for the 16b networks increases from 1.7× up to 3.1× and for the range-aware quantized 8b models is on average



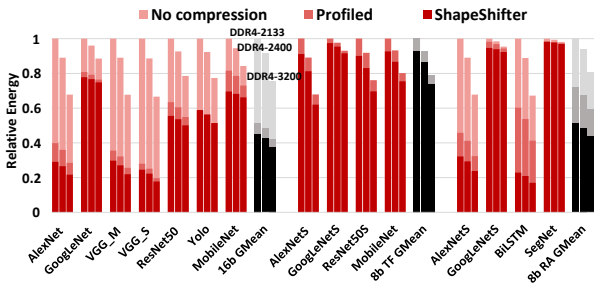
(a) DaDianNao*: Performance



(b) DaDianNao*: Energy



(c) Bit Fusion: Performance



(d) Bit Fusion: Energy

Figure 9: Performance and Energy Efficiency of with *ShapeShifter* compression with DDR4-2133 (left bar), DDR4-2400 (middle bar), and DDR4-3200 (right bar) off-chip memory relative to DDR4-2133 no compression.

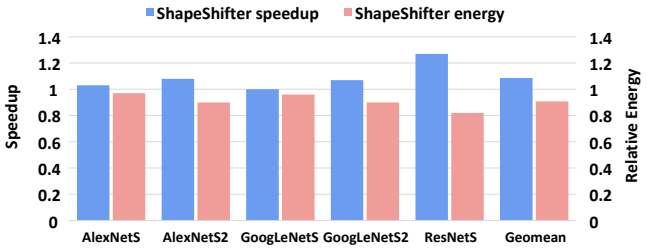


Figure 10: Speedup and Relative Energy for SCNN with *ShapeShifter* and DDR4-2133 off-chip memory normalised to SCNN baseline compression.

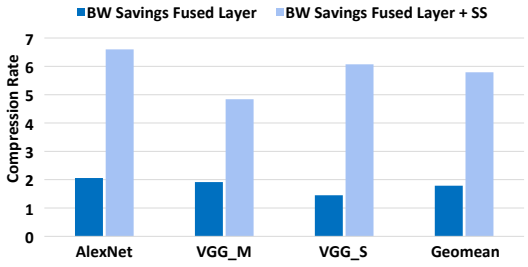


Figure 11: Layer Fusion: Compression ratios with and without *ShapeShifter* as opposed to using neither.

2.93× higher. These benefits require a negligible overhead over Stripes: the detection logic requires just $0.188mm^2$, which is an additional 1.93% (0.3%) in area plus 0.3% (0.06%) power compared to the compute cores (cores+on-chip memory). Similarly, each of the L1D and L2D decoders requires less than 0.02% area.

5.2.1 Comparison with Bit Fusion. We compare *SStripes* to Bit Fusion under iso-area constraints. Bit Fusion uses profiled, per layer precisions for compute and memory compression. We conservatively scaled Stripes to 45nm following constant voltage scaling [69]. Under these constraints Bit Fusion is configured with 512 units per tile. Figure 14 reports relative execution time for *SStripes* over Bit Fusion. Since Bit Fusion suffers from significant time overheads when processing layers using more than 8b we report results for 8b networks only.

In general, *SStripes* is faster than Bit Fusion on these models, with the benefits being more pronounced for the range-aware quantized models, where *SStripes* is 3.75× faster on average. *SStripes* benefits from per group width adaptation and delivers benefits even for precisions that are not power of 2. However, we note that these models are not quantized to better fit Bit Fusion. Unfortunately, it is not currently possible to investigate performance using the models used in the original Bit Fusion study. Only the precision profiles are available for them whereas studying *SStripes* requires access to the values too. Regardless, as we have seen, Bit Fusion benefits from

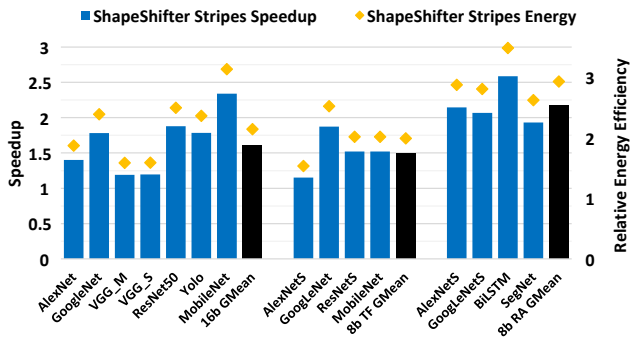


Figure 12: Speedup and Relative Energy Efficiency of *SStripes* over Stripes. Iso-Area Comparison.

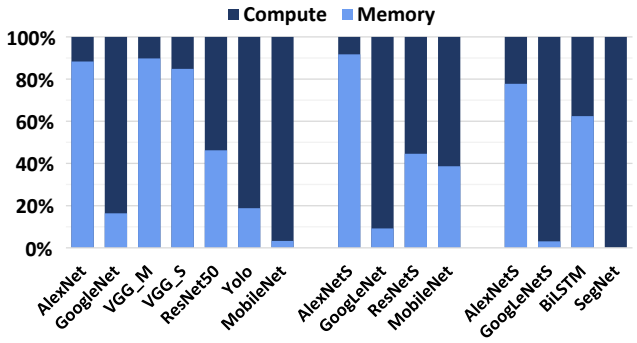


Figure 13: *ShapeShifter* Compute-Memory breakdown.

ShapeShifter compression, and we expect these benefits to persist even for models explicitly quantized for Bit Fusion. Moreover, future work may investigate whether it is possible to modify Bit Fusion to reconfigure its compute units at a finer in time granularity than that of a layer and with the value content.

Figure 14 shows that *SStripes* is also more energy efficient than Bit Fusion with the benefits closely following the performance improvements. The benefits are the lowest for the Tensorflow quantized AlexNetS for two reasons: differences between profiled and dynamic precisions are small, and the model is memory bound due to the its relatively large fully-connected layers.

5.2.2 Smaller On-Chip Memory Configurations. Figure 15 shows how *SStripes* performs with limited on-chip buffers. As on-chip storage diminishes, performance becomes limited by off-chip bandwidth. *SStripes* provides benefit in both regimes.

5.3 ShapeShifter Loom

Dynamic width adaptation can also benefit Loom [70], an architecture that exploits width variability both in activations and weights. Loom is faster and more energy efficient than *Stripes* for smaller configurations. Loom uses bit-serial processing similar to *Stripes*. Due to limited space we report summary results: for the 8b RA models, *ShapeShifter* Loom with 16b SIPs (no composition) performs 2.1× faster on average, and up to 2.3× for GoogLeNetS.

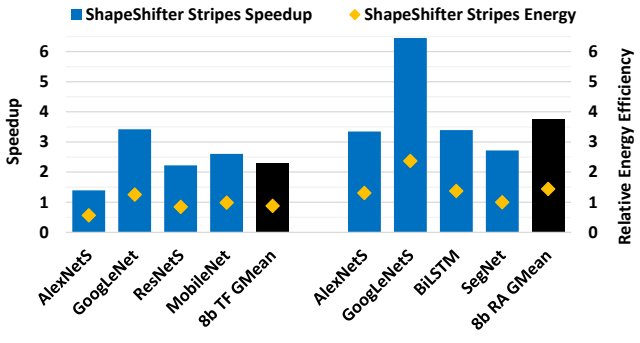


Figure 14: Speedup and Relative Energy Efficiency of *SStripes* over Bit Fusion. Iso-Area comparison.

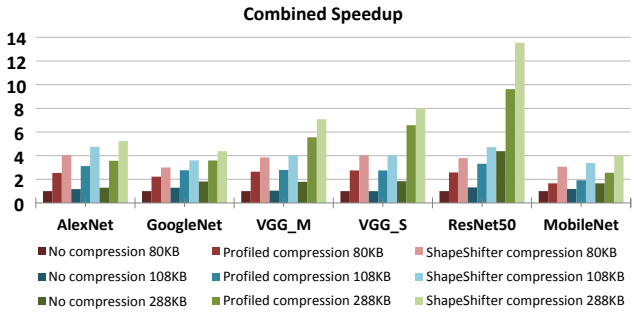


Figure 15: Performance of Stripes using *ShapeShifter* for limited on-chip buffers, with DDR4-3200 external memory.

5.4 Outlier-Aware Quantization

Figure 16 shows the resulting compression of ShapeShifter when applied to Outlier-Aware Quantized networks [32]. MobileNetV2 [62] and pruned ResNet50 [33] are quantized with 1% 16b outliers and 5b and 4b common values respectively. The parameters were chosen to maintain comparable TOP-1 accuracy to full precision [32]. The figure reports memory traffic vs. storing all values in 16b [58].

We compare ShapeShifter with Outlier-Aware and "Outlier-Aware with zero skipping" (ZS) schemes. Outlier Aware stores the common values in 4b or 5b, and the outliers in 32b, 16b for the value and 16 for the position index (using a position index was more space efficient than using an index per group of 16 values). Furthermore, Outlier-Aware ZS compresses zero values to 1b by allocating an additional bit for every non-outlier value. *ShapeShifter* compression outperforms the Outlier-Aware scheme. In comparison to the Outlier Aware ZS scheme, our approach performs better for the dense MobileNetV2 weights, while maintaining similar compression ratios for the sparse weights and activations.

These results demonstrate that *ShapeShifter* delivers off-chip memory benefits with models quantized with outlier-aware quantization. Section 5.1 demonstrated that *ShapeShifter* does the same with two other quantization schemes. Collectively, these demonstrate that *ShapeShifter* effectively delivers most of the memory

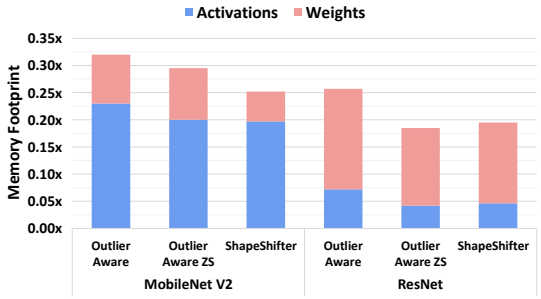


Figure 16: Performance of *SStripes* compression with Outlier-Aware quantized weights and activations.

traffic and footprint reduction benefits possible with various quantization methods *without* mandating that all networks use a specific method.

Demonstrating compute performance benefits with outlier-aware quantized networks and a comparison with accelerator designs specifically tailored for outlier-aware quantized models is left for future work. If all networks of interest can be quantized with the outlier-aware technique, a more specialized design such as that of Park *et al.*, [71] may be preferable.

6 RELATED WORK

The *Efficient Inference Engine* (EIE) uses weight pruning and sharing, zero activation elimination, and network retraining to drastically reduce the weight storage and communication when processing fully-connected layers [72]. This is an aggressive form of weight quantization and compression and where possible will greatly outperform *ShapeShifter* for weights. However, *ShapeShifter* is complementary to the Deep Compression method used by EIE. *ShapeShifter* benefits activations also, works for all layers, and exploits a phenomenon that is mostly orthogonal to those EIE exploits. Moreover, in recent CNNs the size and number of fully-connected layer has been drastically decreased compared to early CNN models. While codebook-based compression methods such as Deep Compression can be applied to convolutional layers, these require retraining and larger codebooks [73].

Datatypes such as Intel's Flexpoint [27] or Tensorflow's bfloat16 [28] are lower precision alternatives to 32-bit floating point. They provide a wider range than fixed-point representations. While they target primarily training there are models, for example in natural language processing or recommendation systems, that achieve their best accuracy only with floating-point arithmetic. These floating-point datatypes are *statically* sized and require 7, 8 or 16 bits of mantissa, and 8 bits of exponent regardless of value magnitude. The models will still exhibit a lopsided value distribution and will natural exhibit variable precision requirements per layer. Accordingly, it may be possible to apply *ShapeShifter* to the mantissa and exponent to adjust the least significant bits used after profile has been used to trim the precision needed per layer.

Quantization to lower precisions reduces memory and computation cost in neural networks. However, quantization is not without its drawbacks. Although a fixed precision may be suitable for certain networks, using a fixed quantization scheme may not be optimal/sufficient for all problems, e.g., those using RNNs [3] or computational imaging tasks performing per-pixel prediction, e.g., [48, 49] which process raw sensor data of 12b or more, or even some image classification tasks and speech processing [74]. Therefore, flexible precision support is presently important to generalize across different applications. While quantization attenuates some of the expected benefits with *ShapeShifter* the underlying phenomenon that *ShapeShifter* exploits persists: there will be few high values. This is the reason why it benefits even 8b quantized networks. Further, *SStripes* offers benefits even for those networks that can be quantized further, even for non powers of two precisions, e.g., [75] without requiring all networks to be quantized to obtain benefits. This applies also to models quantized to binary weights [76] or activations.

The Compressing DMA engine also uses vector-based zero compression [77] and has shown that it is very effective for reducing off-chip traffic and storage during training.

Tartan extends Stripes so it can take advantage of per layer weight precisions [78] so that it can speed up fully-connected layers too. *ShapeShifter* is directly compatible with Tartan and would increase benefits by adjusting precisions per weight group instead. Due to limited space an evaluation of this design is left for future work.

Diffy improves upon *ShapeShifter* by using it to encode activations as deltas [79]. Diffy exploits the spatial value correlation found in the activation values of neural networks implementing computational imaging tasks.

The idea of adapting operand precision for applications exhibiting algorithmic fault tolerance has been proposed by Nam Sung et al. [80], transmitting a fixed number of shift amounts (1-valued bits) per value at the cost of larger accuracy loss.

7 CONCLUSION

Per group precision adaptation opens up several directions for future work including how to combine with other accelerator engines, how it can boost the effectiveness of algorithms for pruning or for precision reduction or quantization of weights and of activations, and whether it can accelerate training. The accelerators we proposed *do not require* any changes to the input network to deliver benefits. However, they *do reward* advances in precision reduction including quantization and thus if deployed will provide incentive for further innovation.

ACKNOWLEDGEMENTS

This work was supported in part by the NSERC COHESA Research Network, an NSERC Discovery Grant, a Canadian DND Discovery Grant Accelerator Supplement, and an NSERC Strategic Grant. Earlier versions of this manuscript [81] have been submitted for peer review at HPCA 2018, ISCA 2018, MICRO 2018, HPCA 2019, and ISCA 2019. We thank the reviewers of the aforementioned conferences and of MICRO 2019 for their comments and time.

REFERENCES

[1] D. Das, N. Mellempudi, D. Mudigere, D. D. Kalamkar, S. Avancha, K. Banerjee, S. Sridharan, K. Vaidyanathan, B. Kaul, E. Georganas, A. Heinecke, P. Dubey,

J. Corbal, N. Shustrov, R. Dubtsov, E. Fomenko, and V. O. Pirogov, "Mixed precision training of convolutional neural networks using integer operations," *CoRR*, vol. abs/1802.00930, 2018.
[2] M. Drumond, T. Lin, M. Jaggi, and B. Falsafi, "End-to-end DNN training with block floating point arithmetic," *CoRR*, vol. abs/1804.01526, 2018.
[3] S. Migacz, "8-bit inference with tensorrt," 2017. GPU Technology Conference.
[4] P. Warden, "Low-precision matrix multiplication." https://petewarden.com, 2016.
[5] M. Courbariaux, Y. Bengio, and J. David, "Low precision arithmetic for deep learning," *CoRR*, vol. abs/1412.7024, 2014.
[6] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, "Deep learning with limited numerical precision," in *Proceedings of the 32Nd International Conference on International Conference on Machine Learning - Volume 37*, ICML'15, pp. 1737–1746, JMLR.org, 2015.
[7] A. K. Mishra, E. Nurvitadhi, J. J. Cook, and D. Marr, "WRPN: wide reduced-precision networks," *CoRR*, vol. abs/1709.01134, 2017.
[8] E. Park, J. Ahn, and S. Yoo, "Weighted-entropy-based quantization for deep neural networks," in *2017 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017, Honolulu, HI, USA, July 21-26, 2017*, pp. 7197–7205, 2017.
[9] S. Zhou, Z. Ni, X. Zhou, H. Wen, Y. Wu, and Y. Zou, "Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients," *CoRR*, vol. abs/1606.06160, 2016.
[10] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, "Quantized neural networks: Training neural networks with low precision weights and activations," *Journal of Machine Learning Research*, vol. 18, pp. 187:1–187:30, 2017.
[11] S. Kapur, A. K. Mishra, and D. Marr, "Low precision rnns: Quantizing rnns without losing accuracy," *CoRR*, vol. abs/1710.07706, 2017.
[12] C. Zhu, S. Han, H. Mao, and W. J. Dally, "Trained ternary quantization," *CoRR*, vol. abs/1612.01064, 2016.
[13] F. Li and B. Liu, "Ternary weight networks," *CoRR*, vol. abs/1605.04711, 2016.
[14] M. Courbariaux, Y. Bengio, and J.-P. David, "BinaryConnect: Training Deep Neural Networks with binary weights during propagations," *ArXiv e-prints*, Nov. 2015.
[15] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, "Xnor-net: Imagenet classification using binary convolutional neural networks," *CoRR*, vol. abs/1603.05279, 2016.
[16] L. Deng, P. Jiao, J. Pei, Z. Wu, and G. Li, "Gxnor-net: Training deep neural networks with ternary weights and activations without full-precision memory under a unified discretization framework," *Neural Networks*, vol. 100, pp. 49–58, 2018.
[17] M. Kim and P. Smaragdis, "Bitwise neural networks," *CoRR*, vol. abs/1601.06071, 2016.
[18] J. Kim, K. Hwang, and W. Sung, "X1000 real-time phoneme recognition VLSI using feed-forward deep neural networks," in *2014 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 7510–7514, May 2014.
[19] B. Moons, R. Uytterhoeven, W. Dehaene, and M. Verhelst, "Envision: A 0.26-to-10tops/w subword-parallel dynamic-voltage-accuracy-frequency-scalable convolutional neural network processor in 28nm fdsoi," in *IEEE Solid-State Circuits Conference (ISSCC)*, 2017.
[20] S. Shin, K. Hwang, and W. Sung, "Fixed point performance analysis of recurrent neural networks," *CoRR*, vol. abs/1512.01322, 2015.
[21] P. Judd, J. Albericio, T. Hetherington, T. Aamodt, N. E. Jerger, R. Urtasun, and A. Moshovos, "Reduced-Precision Strategies for Bounded Memory in Deep Neural Nets, arXiv:1511.05236v4 [cs.LG] ," *arXiv.org*, 2015.
[22] P. Judd, J. Albericio, T. Hetherington, T. M. Aamodt, N. E. Jerger, and A. Moshovos, "Proteus: Exploiting numerical precision variability in deep neural networks," in *Proceedings of the 2016 International Conference on Supercomputing*, ICS '16, (New York, NY, USA), pp. 23:1–23:12, ACM, 2016.
[23] J. Qiu, J. Wang, S. Yao, K. Guo, B. Li, E. Zhou, J. Yu, T. Tang, N. Xu, S. Song, Y. Wang, and H. Yang, "Going deeper with embedded fpga platform for convolutional neural network," in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '16, (New York, NY, USA), pp. 26–35, ACM, 2016.
[24] D. D. Lin, S. S. Talathi, and V. S. Annapureddy, "Fixed point quantization of deep convolutional networks," in *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48*, ICML'16, pp. 2849–2858, JMLR.org, 2016.
[25] H. Sharma, J. Park, N. Suda, L. Lai, B. Chau, V. Chandra, and H. Esmaeilzadeh, "Bit fusion: Bit-level dynamically composable architecture for accelerating deep neural network," in *ISCA*, pp. 764–775, IEEE Computer Society, 2018.
[26] Z. Song, Z. Liu, and D. Wang, "Computation error analysis of block floating point arithmetic oriented convolution neural network accelerator design," in *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018*, pp. 816–823, 2018.
[27] U. Köster, T. Webb, X. Wang, M. Nassar, A. K. Bansal, W. Constable, O. Elibol, S. Hall, L. Hornof, A. Khosrowshahi, C. Kloss, R. J. Pai, and N. Rao, "Flexpoint: An adaptive numerical format for efficient training of deep neural networks," in *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, 4-9 December 2017, Long Beach, CA, USA*, pp. 1740–1750, 2017.
[28] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015. Software available from tensorflow.org.
[29] P. Judd, J. Albericio, T. Hetherington, T. Aamodt, and A. Moshovos, "Stripes: Bit-serial Deep Neural Network Computing ," in *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-49, 2016.
[30] S. Sharify, A. D. Lascorz, P. Judd, and A. Moshovos, "Loom: Exploiting weight and activation precisions to accelerate convolutional neural networks," *CoRR*, vol. abs/1706.07853, 2017.
[31] C. Eckert, X. Wang, J. Wang, A. Subramaniyan, R. R. Iyer, D. Sylvester, D. T. Blaauw, and R. Das, "Neural cache: Bit-serial in-cache acceleration of deep neural networks," in *45th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2018, Los Angeles, CA, USA, June 1-6, 2018*, pp. 383–396, 2018.
[32] E. Park, D. Kim, and S. Yoo, "Energy-efficient neural network accelerator based on outlier-aware low-precision computation," in *ISCA*, pp. 688–698, IEEE Computer Society, 2018.
[33] J. Park, S. Li, W. Wen, P. T. P. Tang, H. Li, Y. Chen, and P. Dubey, "Faster CNNs with Direct Sparse Convolutions and Guided Pruning," in *5th International Conference on Learning Representations (ICLR)*, 2017.
[34] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, "ImageNet Large Scale Visual Recognition Challenge," *arXiv:1409.0575 [cs]*, Sept. 2014. arXiv: 1409.0575.
[35] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," *arXiv preprint arXiv:1408.5093*, 2014.
[36] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *CoRR*, vol. abs/1512.03385, 2015.
[37] Yang, Tien-Ju and Chen, Yu-Hsin and Sze, Vivienne, "Designing Energy-Efficient Convolutional Neural Networks using Energy-Aware Pruning," in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017.
[38] F. N. Iandola, M. W. Moskewicz, K. Ashraf, S. Han, W. J. Dally, and K. Keutzer, "Squeezenet: Alexnet-level accuracy with 50x fewer parameters and <1mb model size," *CoRR*, vol. abs/1602.07360, 2016.
[39] 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," *CoRR*, vol. abs/1704.04861, 2017.
[40] G. J. Brostow, J. Fauqueur, and R. Cipolla, "Semantic object classes in video: A high-definition ground truth database," *Pattern Recognition Letters*, vol. xx, no. x, pp. xx–xx, 2008.
[41] V. Badrinarayanan, A. Kendall, and R. Cipolla, "Segnet: A deep convolutional encoder-decoder architecture for image segmentation," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2017.
[42] M. Everingham, S. M. A. Eslami, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman, "The pascal visual object classes challenge: A retrospective," *International Journal of Computer Vision*, vol. 111, pp. 98–136, Jan. 2015.
[43] J. Redmon and A. Farhadi, "YOLO9000: better, faster, stronger," *CoRR*, vol. abs/1612.08242, 2016.
[44] E. Shelhamer, J. Long, and T. Darrell, "Fully convolutional networks for semantic segmentation," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 39, pp. 640–651, April 2017.
[45] S. Roth and M. J. Black, "Fields of experts: a framework for learning image priors," in *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*, vol. 2, pp. 860–867 vol. 2, June 2005.
[46] M. Bevilacqua, A. Roumy, C. Guillemot, and M. Alberi-Morel, "Low-complexity single-image super-resolution based on nonnegative neighbor embedding," in *British Machine Vision Conference, BMVC 2012, Surrey, UK, September 3-7, 2012*, pp. 1–10, 2012.
[47] R. Zeyde, M. Elad, and M. Protter, "On single image scale-up using sparse-representations," in *Curves and Surfaces* (J.-D. Boissonnat, P. Chenin, A. Cohen, C. Gout, T. Lyche, M.-L. Mazure, and L. Schumaker, eds.), (Berlin, Heidelberg), pp. 711–730, Springer Berlin Heidelberg, 2012.
[48] K. Zhang, W. Zuo, S. Gu, and L. Zhang, "Learning deep cnn denoiser prior for image restoration," in *IEEE Conference on Computer Vision and Pattern Recognition*, pp. 3929–3938, 2017.
[49] D. Li and Z. Wang, "Video superresolution via motion compensation and deep residual learning," *IEEE Transactions on Computational Imaging*, vol. 3, pp. 749–762, Dec 2017.
[50] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," in *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'14, (Cambridge, MA, USA),

pp. 3104–3112, MIT Press, 2014.
[51] T. Lin, M. Maire, S. J. Belongie, L. D. Bourdev, R. B. Girshick, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick, "Microsoft COCO: common objects in context," *CoRR*, vol. abs/1405.0312, 2014.
[52] J. Donahue, L. A. Hendricks, S. Guadarrama, M. Rohrbach, S. Venugopalan, K. Saenko, and T. Darrell, "Long-term recurrent convolutional networks for visual recognition and description," in *CVPR*, 2015.
[53] C. Rashtchian, P. Young, M. Hodosh, and J. Hockenmaier, "Collecting image annotations using amazon's mechanical turk," in *Proceedings of the NAACL HLT 2010 Workshop on Creating Speech and Language Data with Amazon's Mechanical Turk*, CSLDAMT '10, (Stroudsburg, PA, USA), pp. 139–147, Association for Computational Linguistics, 2010.
[54] C. Wang, H. Yang, C. Bartz, and C. Meinel, "Image captioning with deep bidirectional lstms," in *Proceedings of the 2016 ACM on Multimedia Conference*, pp. 988–997, ACM, 2016.
[55] D. Clevert, T. Unterthiner, and S. Hochreiter, "Fast and accurate deep network learning by exponential linear units (elus)," *CoRR*, vol. abs/1511.07289, 2015.
[56] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," in *2015 IEEE International Conference on Computer Vision (ICCV)*, pp. 1026–1034, Dec 2015.
[57] V. Young, S. Kariyappa, and M. K. Qureshi, "Enabling transparent memory-compression for commodity memory systems," in *25th IEEE International Symposium on High Performance Computer Architecture, HPCA 2019, Washington, DC, USA, February 16-20, 2019*, pp. 570–581, 2019.
[58] K. Siu, D. M. Stuart, M. Mahmoud, and A. Moshovos, "Memory requirements for convolutional neural network hardware accelerators," in *IEEE International Symposium on Workload Characterization*, 2018.
[59] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems 25: 26th Annual Conference on Neural Information Processing Systems 2012. Proceedings of a meeting held December 3-6, 2012, Lake Tahoe, Nevada, United States.*, pp. 1106–1114, 2012.
[60] K. Chatfield, K. Simonyan, A. Vedaldi, and A. Zisserman, "Return of the devil in the details: Delving deep into convolutional nets," *CoRR*, vol. abs/1405.3531, 2014.
[61] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *CoRR*, vol. abs/1512.03385, 2015.
[62] M. Sandler, A. G. Howard, M. Zhu, A. Zhmoginov, and L. Chen, "Inverted residuals and linear bottlenecks: Mobile networks for classification, detection and segmentation," *CoRR*, vol. abs/1801.04381, 2018.
[63] Synopsys, "Design Compiler." http://www.synopsys.com/Tools/Implementation/RTLSynthesis/DesignCompiler/Pages.
[64] N. Muralimanohar and R. Balasubramonian, "Cacti 6.0: A tool to understand large caches."
[65] X. Yang, M. Gao, J. Pu, A. Nayak, Q. Liu, S. Bell, J. Setter, K. Cao, H. Ha, C. Kozyrakis, and M. Horowitz, "DNN dataflow choice is overrated," *CoRR*, vol. abs/1809.04070, 2018.
[66] S. Han, H. Mao, and W. J. Dally, "Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding," *arXiv:1510.00149 [cs]*, Oct. 2015. arXiv: 1510.00149.
[67] M. Alwani, H. Chen, M. Ferdman, and P. Milder, "Fused-layer cnn accelerators," in *49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016.
[68] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-l. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, 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, M. 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," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, (New York, NY, USA), pp. 1–12, ACM, 2017.
[69] N. Weste and D. Harris, *CMOS VLSI Design: A Circuits and Systems Perspective*. USA: Addison-Wesley Publishing Company, 4th ed., 2010.
[70] S. Sharify, A. D. Lascorz, K. Siu, P. Judd, and A. Moshovos, "Loom: Exploiting weight and activation precisions to accelerate convolutional neural networks," in *Proceedings of the 55th Annual Design Automation Conference*, DAC '18, (New York, NY, USA), pp. 20:1–20:6, ACM, 2018.
[71] E. Park, D. Kim, and S. Yoo, "Energy-efficient neural network accelerator based on outlier-aware low-precision computation," in *ISCA*, pp. 688–698, IEEE Computer Society, 2018.
[72] 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," in *43rd ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2016, Seoul, South Korea, June 18-22, 2016*, pp. 243–254, 2016.
[73] X. Zhou, Z. Du, Q. Guo, C. Liu, C. Wang, X. Zhou, L. Li, T. Chen, and Y. Chen, "Cambricon-s: Addressing irregularity in sparse neural networks through a cooperative software/hardware approach," in *Proceedings of the 51th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO*, 2018.
[74] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. G. Howard, H. Adam, and D. Kalenichenko, "Quantization and training of neural networks for efficient integer-arithmetic-only inference," *CoRR*, vol. abs/1712.05877, 2017.
[75] E. Park, S. Yoo, and P. Vajda, "Value-aware quantization for training and inference of neural networks," in *Computer Vision - ECCV 2018 - 15th European Conference, Munich, Germany, September 8-14, 2018, Proceedings, Part IV*, pp. 608–624, 2018.
[76] M. Courbariaux, Y. Bengio, and J.-P. David, "Binaryconnect: Training deep neural networks with binary weights during propagations," in *Advances in Neural Information Processing Systems*, pp. 3123–3131, 2015.
[77] 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 *IEEE International Symposium on High Performance Computer Architecture, HPCA 2018, Vienna, Austria, February 24-28, 2018*, pp. 78–91, 2018.
[78] A. Delmas, S. Sharify, P. Judd, and A. Moshovos, "Tartan: Accelerating fully-connected and convolutional layers in deep learning networks by exploiting numerical precision variability," *CoRR*, vol. abs/1707.09068, 2017.
[79] M. Mahmoud, K. Siu, and A. Moshovos, "Diffy: A dÉjÀ vu-free differential deep neural network accelerator," in *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-51, (Piscataway, NJ, USA), pp. 134–147, IEEE Press, 2018.
[80] N. Kim, T. Park, S. Narayanamoorthy, and H. Asgharimoghaddam, "Multiplier supporting accuracy and energy trade-offs for recognition applications," *IET Electronics Letters*, vol. 50, no. 7, pp. 512–514, 2014.
[81] A. Delmas, S. Sharify, P. Judd, M. Nikolic, and A. Moshovos, "Dpred: Making typical activation values matter in deep learning computing," *CoRR*, vol. abs/1804.06732, 2018.