TensorDash: Exploiting Sparsity to Accelerate Deep Neural Network Training

Mostafa Mahmoud
University of Toronto
mostafa.mahmoud@mail.utoronto.ca

Isak Edo
University of Toronto
edoisak@ece.utoronto.ca

Ali Hadi Zadeh
University of Toronto
hadizade@ece.utoronto.ca

Omar Mohamed Awad
University of Toronto
awadomar@ece.utoronto.ca

Gennady Pekhimenko
University of Toronto, Vector Institute
pekhimenko@cs.toronto.edu

Jorge Albericio
Cerebras Systems
jorge@cerebras.net

Andreas Moshovos
University of Toronto, Vector Institute
moshovos@ece.utoronto.ca

***Abstract*—*TensorDash* is a hardware-based technique that enables data-parallel MAC units to take advantage of sparsity in their input operand streams. When used to compose a hardware accelerator for deep learning, *TensorDash* can speedup the *training* process while also increasing energy efficiency. *TensorDash* combines a low-cost sparse input operand interconnect with an area-efficient hardware scheduler. The scheduler can effectively extract sparsity in the activations, the weights, and the gradients. Over a wide set of state-of-the-art models covering various applications, *TensorDash* accelerates the training process by $1.95\times$ while being $1.5\times$ more energy efficient when incorporated on top of a *Tensorcore*-based accelerator at less than 5% area overhead. *TensorDash* is datatype agnostic and we demonstrate it with IEEE standard mixed-precision floating-point units and a popular optimized for machine learning floating-point format (BFloat16).**

I. Introduction

Whereas a decade ago the then state-of-the-art neural networks could be trained on a commodity server within a few hours, today training the best neural networks has become an exascale class problem [6], [69]. State-of-the-art neural networks now require many graphics processors [2] or specialized accelerators such as the TPU [39], Gaudi [4], DaVinci [45], or Cerebras CS1 [3] so that they can be trained within practical time limits. Tuning neural networks, e.g., via hyperparameter exploration [66] or more generally via network architecture search [22], for best performance or accuracy during inference further exacerbates the cost of training. Beyond the cost of acquiring or getting access to such expensive computing resources, worse are the operating costs and the environmental impact of training. Strubell et al., report that the CO_2 emissions of training even a mid-class neural network stand at about 36 metric tons [67]. Training neural networks at the "edge" is needed in certain applications as well, e.g., to refine an existing model with user-specific information and input. While the trade offs for edge devices are different than those for data centers or desktop machines, the need remains the same: reduce execution time and improve energy efficiency albeit under different constraints.

It comes then as no surprise that efforts for reducing the execution time and the energy cost of training have been considerable. Distributed training partitions the training workload across several computing nodes by exploiting model, data, and pipeline parallelism to reduce overall latency [17], [50]. Intra- and inter-node data blocking, reuse, and communication and computation overlapping orchestrate computing, memory hierarchy, and communication resources to improve performance and energy efficiency [10], [35], [74]. Lossless and lossy compression reduces the footprint of the vast amounts of data processed during training [23], [38], [81]. While originally training used single precision floating-point data and arithmetic, more compact datatypes reduce overall data volumes and computation costs (e.g., half precision floating-point FP16, bfloat16 [26], [41], [73], dynamic floating-point [14], and flexpoint [42]). Mixed-datatype methods further reduce costs by performing many computations using lower cost representations and few using higher cost ones [14], [21], [52], [56]. Other methods use low precision arithmetic [16].

TABLE I: DNN models studied.

Application	Model / Abbreviation
Natural language modeling	BERT [20] / BERT
	Stanford Natural Language Inf. [7] / SNLI
Object detection/segmentation	Detectron2 [75] / Dtctron2
Image classification	SqueezeNet [37] / SQZNet
	VGG [65] / VGG16
	ResNet-50 [32] / RsNt50_S
Scene understanding	Show and Tell [70] / img2txt
Recommendation systems	Neural Collaborative Filtering [33] / NCF

Regardless, training remains an exascale class problem and further improvements are needed. We observe that during training many ineffectual computations occur *naturally* and for a variety of models. Table I lists the ones we study. Specifically, the bulk of energy consumption during training is due to the transfers and computations needed to perform multiply-accumulate operations (MACs). We find that often one of the operands in these MACs is zero, and hence these operations can be safely eliminated as they do not affect the values produced during training and thus convergence and final accuracy. We find that for *many* networks many zeros naturally occur in the activations during the forward and backward passes, and in the gradients during the backward pass (see Section II-A for a primer on training).

It is well-known that zero values and ineffectual operations

978-1-7281-7383-2/20/$31.00 ©2020 IEEE
DOI 10.1109/MICRO50266.2020.00069

occur also during inference both in the weights and the activations. While some zero weights appear naturally, their occurrence can be greatly amplified through pruning [28], [29], [30], [43]. Zero activations also occur naturally during inference and are most frequent in models that use the Rectifier Linear Unit (ReLU) activation function. This has led to many software and hardware proposals for exploiting zeros, the presence of which is referred to as *sparsity* [5], [11], [18], [25], [27], [40], [48], [60], [79], [82]. Many designs target sparsity in the weights, and some target sparsity in both activations and weights [18], [40], [58], [60].

However, exploiting sparsity during training is more challenging than it is for inference. First, just because zeros occur during inference does not imply they should also appear during training. Training starts with some random initialization of the weights, and proceeds to slowly adjust them until the network converges. Eventually, some of the weights will become zeros, but how fast this will occur is not known neither is whether they will stay at zero. Second, the position of zero weights during inference is known and does not change, hence the sparsity pattern is *static*. As a result, for inference we can pre-schedule the computation to best take advantage of the sparsity in weights. This is not the case during training where the weight values keep changing and hence the sparsity pattern is *dynamic*. This pattern varies with every sample and batch in the training dataset and also varies over time. Third, inference involves two input tensors, the weights and the activations, which are used in only one computation, typically a matrix-matrix multiplication or a matrix-vector multiplication. Thus, the two tensors can be laid out in memory in a way that serves a specific access pattern facilitating data parallel, and thus energy-efficient, fetching and execution. During training there is a third tensor, the gradients, and each of the three tensors is used in two different computations. Most challenging is that the way a tensor is used in each of those two computations is different. For example, during the *forward* pass, a different set of weights contribute to an output than those during the *backward* pass. This makes it hard to layout the values in memory in a way that fits both computation needs; a layout that fits the forward has to be "transposed" for the backward. Fourth, most inference accelerators that exploit sparsity operate on fixed-point values, whereas training typically requires floating point values. The relative costs of operations is different and may result in different tradeoffs.

When sparsity exists, it represents an opportunity for improving performance and energy efficiency. To exploit this opportunity, we develop a method that will achieve this when sparsity exists, and also avoid hurting performance and energy efficiency otherwise. We present *TensorDash*, a run-time approach to eliminate ineffectual MACs using a combination of an inexpensive hardware scheduler and a co-designed sparse, low-cost data interconnect that are placed just in front of the MAC units. *TensorDash* works with out-of-the-box neural networks and requires no modification nor any special annotations from the model developer. *TensorDash* does not change the values nor the functional units in any way and thus does not affect convergence nor accuracy.

TensorDash also extracts additional benefits from techniques that perform network pruning and quantization during training. The goal of pruning is to convert weight values to zero. Dynamic sparse reparameterization [54], sparse momentum [19], eager pruning [78] and DropBack [24] are recent training-time pruning methods that achieve high sparsity levels with minimal or no effect on output accuracy. We study the interaction of *TensorDash* with some of these methods. Quantization reduces the data width that will be used during inference. During training, quantization effectively clips what would otherwise be values of low magnitude into zeros. Recent quantization methods include PACT [13] and LQ-Nets [77]. *TensorDash* would also benefit selective backpropagation methods which backpropagate loss only for some of the neurons [68]. Unless specialized hardware is developed, selective backpropagation manifests as sparsity as it effectively converts a large number of gradients into zeros.

Our contribution is that we propose *TensorDash* with the following functionality and benefits:

- *TensorDash* exploits naturally occurring sparsity during training which appears predominantly in the activations and the gradients. Sparsity is exploited dynamically and completely in hardware using a low-overhead hardware scheduler to advance MAC operations in time (earlier cycle) and space (another MAC unit) so that overall computation finishes earlier. The scheduler makes no assumptions about how sparsity is distributed so that it can efficiently handle the dynamic sparsity patterns that arise during training.
- *TensorDash* does not affect numerical fidelity. It only eliminates MAC operations where at least one of the inputs is zero.
- *TensorDash* is compatible with data-parallel processing elements that perform multiple MAC operations all accumulating into a single output and is compatible with any dataflow for such processing elements.
- Benefits with *TensorDash* are amplified with training algorithms that incorporate quantization or pruning.
- The core processing element *TensorDash* uses can be configured to extract sparsity in one or both operands. For training, we configure it to do so only on one side as this proves sufficient.

We highlight the following experimental observations:

- When incorporated into an accelerator based on *Tensorcore* (*TC*) processing units, *TensorDash* improves performance by 1.95× and energy efficiency by 1.5× (1.8× for compute units) on average over a set of deep learning models covering a wide range of applications.
- Performance improvements with *TensorDash* remain stable throughout the training process.
- The area overhead of *TensorDash* is 4.8% compared to the TC baseline.
- For bfloat16 units, the area overhead of *TensorDash* is still affordable at 6.5% compared to the baseline.

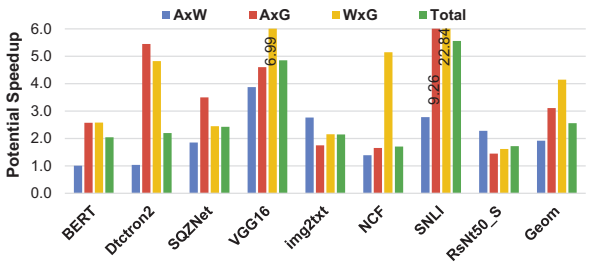


Fig. 1: Potential speedup for exploiting dynamic sparsity during training for each of the three convolutions.

- The cost vs. performance tradeoff with *TensorDash* is better for wider processing units (16 MACs/cycle).

II. Background and Motivation

For clarity we discuss convolutional layers only as their dataflow is more challenging than other layers due to the use of activation windows. However, our measurements include *all* layers. During training, processing a layer i comprises three main convolutions or operations:

$$forward\ pass: \quad A_{i+1} = W_i \star A_i \tag{1}$$

$$backward\ pass: \quad G_{i-1} = G_i \star W_i \tag{2}$$

$$backward\ pass: \quad Gw_i = G_i \star A_i \tag{3}$$

where W_i are the weights, A_i are the input activations, A_{i+1} are the output activations, G_{i-1} are the input activation gradients, G_i are the output activation gradients, and Gw_i are the weight gradients. The first convolution is done during the forward pass to calculate the output activations of the layer while the next two convolutions are done during the back-propagation pass to calculate the input gradients and the weight gradients respectively. Section II-A reviews these operations in more detail. Prior works [38], [62] have demonstrated that the activations of convolutional neural networks exhibit significant sparsity during training and exploited this observation to compress the zeros away off-chipThis section corroborates these findings and shows what levels of sparsity exist in the three operations. Our goal is to exploit sparsity to accelerate the processing by eliminating the corresponding MAC operations.

We found that weights exhibit negligible sparsity during training unless the training method incorporates pruning. However, we observe considerable sparsity in the activations and the output gradients. Thus, we consider exploiting the sparsity of A_i and G_i in the first and the second convolutions respectively. For the third convolution, we target sparsity in G_i or A_i whichever is higher. This can be decided by measuring the fraction of zeros in each tensor as it is being generated by the previous layer. The mechanisms we propose can exploit sparsity for both G_i and A_i simultaneously but we leave the evaluation of this option for future work.

Fig. 1 reports the total potential work reduction and for each of the three operations per model (see Section IV for the methodology behind this and other experiments). The forward propagation operation ($A \times W$) and the two backward propagation operations ($A \times G$) and ($W \times G$) each performs roughly the same number of MACs. We report work reduction as a speedup which we define as $\frac{all\ MACs}{remaining\ MACs}$ where *remaining MACs* is the number of MAC operations left after eliminating those where the targeted operand is zero. On average across all models, the potential "speedup" for the convolutions is nearly 2.6×. The least potential is exhibited by the Neural Collaborative Filtering (*NCF*) recommendation system but even there it is 1.7×. The potential can go as high as 5.6× for the natural language inference model SNLI. It is more than 2.4× for the highly optimized SqueezeNet while being 2.2× for Facebook's object detection and segmentation model Detectron2. For BERT the potential is 2×. While ResNet50 is originally a dense model, pruning techniques induce significant sparsity raising the potential to 1.75×.

A. Training Basics

Deep neural networks are trained using a variant of the gradient descent algorithm where training samples are run through the network to find the prediction error (gradients) relative to the corresponding labels (forward pass) and then the gradients are back-propagated through the network layers to update the network parameters (backward pass).

During the forward pass, $A \star W$ is applied in sequence from the first to the last layer. At every layer it convolves the weights with the input activations to produce the output activations to be fed to the next layer. The output activations of the very last layer are compared with the ground truth labels to generate the gradients that will then be back-propagated to update the weights throughout. During back-propagation the layers are invoked in reverse order from the last to the first. Each layer convolves its output gradients with the weights to produce the input gradients to be fed to the preceding layer. The layer also convolves its output gradients with its input activations to calculate the weight gradients. The per layer weight gradients are accumulated across the training samples within a mini-batch and used to update the weights once per mini-batch, or iteration, as described by Equation Eq. (10), where i is the layer number, t is the iteration number, α is the learning rate, and S is the mini-batch size.

$$W_i^{t+1} = W_i^t - \alpha * \sum_{s=0}^{S} Gw_i^s / S \tag{10}$$

Table II describes the operations in more detail for a convolutional layer and a fully-connected layer, which can be treated as a special-case convolutional layer where all input tensors are of equal size.

III. Exploiting Sparsity: Training vs. Inference

A popular architecture for training are NVIDIA GPUs with the *Tensorcore* extension (*TCs*) [2]. Thus, we assume the main building block of our baseline accelerator to be *Tensorcore*-like units mirroring the functionality of the NVIDIA Volta V100 [2]. The TC is designed to maximize computation throughput under the data supply constraints of the existing memory datapath and, as a result, the internal organization may vary per GPU generation. A TC in V100 can perform a 4×4 floating-point

TABLE II: Training Process: Processing of one training sample. Weights are updated per batch (see text). The notation used for activations, weights, activation gradients, weight gradients is respectively $A^{S/L}_{c,x,y}, W^{L,F}_{c,x,y}, G^{S/L}_{c,x,y}, Gw^{S/L,F}_{c,x,y}$, where S refers to the training sample, L refers to the network layer, F is the weight filter, c is the channel number, and x, y are the 2D spatial coordinates. The stride is denoted as st.

FORWARD PASS

Convolutional Layer: A sliding-window 3D convolution is performed between the input activations and each of the weight filers to produce one channel in the output activations:

$$A^{S/i+1}_{oc,ox,oy} = \sum_{ci=0}^{C} \sum_{xi=0}^{Kx} \sum_{yi=0}^{Ky} A^{S/i}_{ci,ox*st+xi,oy*st+yi} * W^{i,oc}_{ci,xi,yi} \quad (4)$$

Fully-Connected: Each filter produces one output activation:

$$A^{S/i+1}_{oc} = \sum_{ci=0}^{C} A^{S/i}_{ci} * W^{i,oc}_{ci} \quad (5)$$

BACKWARD PASS

INPUT GRADIENTS

Convolutional Layer: A sliding-window 3D convolution is performed between a *reshaped* version of the filters with the activation gradients from the subsequent layer. The filters are reconstructed channel-wise and rotated by 180 degrees and the activation gradients are dilated by the stride st.

$$G^{S/i-1}_{oc,ox,oy} = \sum_{ci=0}^{F} \sum_{xi=0}^{Kx} \sum_{yi=0}^{Ky} G^{S/i}_{ci,ox+xi,oy+yi} * W_{rotated}{}^{i,ci}_{oc,xi,yi} \quad (6)$$

Fully-Connected: The filters are reconstructed as above. No dilation of the activation gradients.

$$G^{S/i-1}_{oc} = \sum_{ci=0}^{F} G^{S/i}_{ci} * W^{i,ci}_{oc} \quad (7)$$

WEIGHT GRADIENTS

Convolutional Layer: The weight gradients are accumulated across batch samples. Per sample, it is calculated as a 2D convolution between the input activation and the output gradients which are dilated according to the stride.

$$Gw^{total/i,f}_{oc,ox,oy} = \sum_{si=0}^{S} \sum_{xi=0}^{Nox} \sum_{yi=0}^{Noy} G^{si/i}_{f,xi,yi} * A^{si/i}_{oc,ox+xi,oy+yi} \quad (8)$$

Fully-Connected: Each weight gradient is a scalar product of the input activation and the gradient of the output activation it affects accumulated over samples.

$$Gw^{total/i,f}_{oc} = \sum_{si=0}^{S} G^{si/i}_{f} * A^{si/i}_{oc} \quad (9)$$

matrix multiplication per cycle, i.e., 64 MACs per cycle. It can be implemented as a tile of 4×4 processing elements (*PEs*) where each PE, as shown in Fig. 2, can perform 4 MACs/cycle all contributing to the same output. For example, these could be four pairs of (activation, weight) all contributing to the same output activation, or they could be four pairs of (gradient, weight) all contributing to the same activation gradient. Such processing elements are more energy efficient

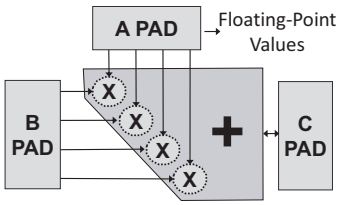


Fig. 2: Baseline Processing Element. A *Tensorcore* is 4×4 tile of PEs.

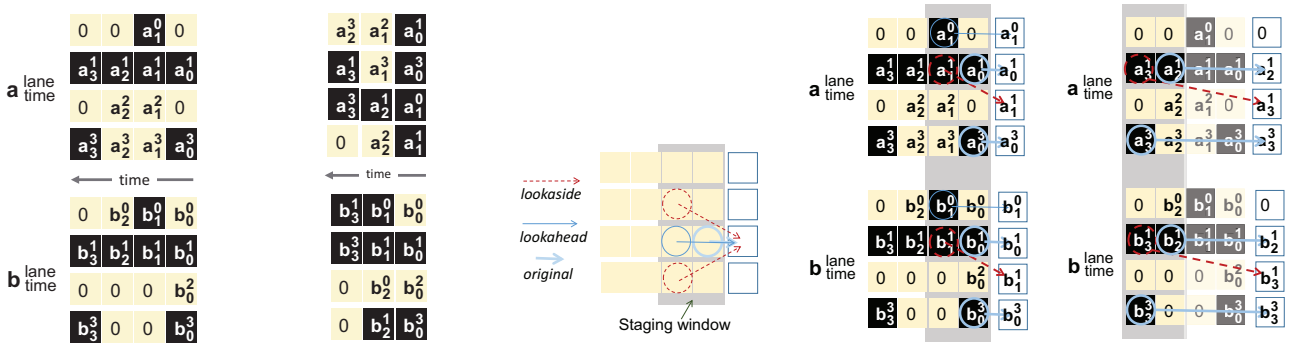
vs. a single MAC unit because they amortize the energy cost of updating the accumulator over several operations, and the cost of the summation stage by fusing the MACs. Similar to TCs, the PEs implement mixed-precision MAC arithmetic where multiplication is done in FP16 while accumulation is performed in FP32. The processing element has three local scratchpads, two for inputs and one for output. An accelerator may use a grid of these PEs each with separate scratchpads or it may organize several of them in a tiled grid sharing the buffers to exploit temporal and spatial reuse. While we assume mixed-precision floating point values as in TCs, *TensorDash* is datatype-agnostic and will work with any datatype, e.g., bfloat16 [41], fixed-point or specialized narrow floating-point [72].

Let us refer to the two input streams as A and B while using C to refer to the outputs. Figure 3a shows an example of how 16 value-pairs will be processed when we do *not* attempt to eliminate those that are *ineffectual* (at least one of the two input values is zero). For non-zero values, we denote the input values as a^{lane}_{time} and b^{lane}_{time}, where *lane* designates the multiplier they appear at, and *time* is the processing order. For zero values, we just have 0. The figure shows that with the *dense schedule*, when we process all pairs regardless of their value, it is straightforward to arrange them in memory so that the PE can read them as groups of four pairs from the input buffers performing four MACs per cycle. The PE needs four cycles to process them all.

In the example, however, there are only seven pairs, highlighted in black, where both operands are non-zeros. As long as the PE processes these value pairs, the output will be correct. To improve performance and to reduce energy, *TensorDash*'s goal is to eliminate the ineffectual pairs by filling their positions with effectual pairs. Ideally, our four MACs/cycle PE should be able to process all effectual pairs in two cycles. However, this requires moving pairs in tandem within both buffers in *time* (earlier yet to the same lane) and in *space-time* (earlier and to a different lane).

To exploit sparsity, we get some inspiration from past designs that did so for *inference* alone [18], [27], [48], [58], [79], [82]. However, as we briefly discussed in the introduction, training is a more challenging task necessitating a fundamentally different approach and design. To appreciate the difference in both challenges and solutions, we first review past approaches for exploiting sparsity during inference.

Inference executes only the $A \star W$ convolution where the weights and their sparsity pattern are known *a priori* . Second, since there is only one convolution and one pass, a single



(a) Input Tensors (b) Unrestricted Movement (c) Sparse Interconnect (d) Cycle 1 (e) Cycle 2

Fig. 3: Example of exploiting sparsity dynamically. Allowing a restricted set of movements per multiplier is sufficient.

dataflow is sufficient so that we can arrange values in memory in the order we wish to process them. The most challenging layers are the convolutional ones, since they use sliding windows in their activations. This means that weights will have to be matched with different activations per window. Fig. 3b shows an approach representative of several past designs where the non-zero values from both sides were allowed to *independently* move with no restriction both in time and space-time [48], [58], [79]. The non-zero values in A are now tightly packed one after the other in memory space, same for B. The values belonging to the same pair are no longer aligned in time nor in space. To avoid processing all ineffectual pairs, we need to somehow identify those pairs where both values are non-zero and make them meet at some multiplier. We would also like to keep as many multipliers busy as possible. This is a challenging task for two reasons: 1) Performing arbitrary movement of values in time and space is expensive in hardware. 2) To keep the four multiplier lanes busy, we will often need to take values from multiple rows from each buffer.

Cambricon is a representative example from a class of designs that exploit sparsity *only* on the weight side [48]. Cambricon tightly packs the non-zero weights in memory space, so that at runtime the PE can access them one row at a time. Each weight is annotated with metadata so that Cambricon can determine its dense $(lane, time)$ position. A unit maintaining a pool of activation candidates locates and pairs each non-zero weight with its activation. This unit proved to be expensive as it performs the function of a crossbar, so that activations can mirror the arbitrary movement of weights in memory space. Cambricon-X exploits sparsity on *both* sides allowing weights and activations to freely move both in time and space-time. An indexing module is tasked with matching non-zero weights and activations [80]. Cambricon-S improves efficiency by imposing *structural* constraints on how the model is pruned [82]. Effectively, it eliminates ineffectual pairs only if 16 of them appear together in a single row. These structural constraints must be imposed during pruning. Cnvlutin2 [40] and SparTen [25] exploit sparsity on both sides albeit by paying the deployment cost for independent buffer banks per multiplier input (both A and B). They support movement of values only in time and hence cannot effectively handle work imbalance across lanes where "Struggler" lanes become a bottleneck. SCNN tightly packs non-zero weights and activations in memory and processes only *effectual* pairs where both operands are not zero at runtime. To do so, it processes values one channel at a time so that the product of any weight with any activation is guaranteed to contribute to an output activation assuming a stride of 1. SCNN avoids all data movement at the input. However, it does require a crossbar to route products to accumulator banks. The crossbar and number of banks are over-provisioned to avoid stalls due to bank conflicts which would otherwise be significant. Bit-Tactical uses a low-cost sparse interconnect at the front-end and a software scheduler to extract sparsity in the weights of pruned models without imposing any restrictions on how sparsity is structured [18]. On the activation side, it targets sparsity within values (bit-level sparsity) and for that it uses shift-and-add multiplier-based MAC units. MASR extracts sparsity from both weights and activations using an efficient-vector based encoding in memory [27]. It targets recurrent neural networks and operates on fixed-point values.

None of the above approaches has been applied in *training*. We highlight the following major challenges that were not addressed by past approaches: 1) The sparsity pattern during training is always *dynamic*. During inference, however, the weights are statically known and, as a result, they can be pre-packed in memory after eliminating zero weights. 2) During training, each tensor participates in two convolutions or operations. The group of values that contribute to an output in each convolution is different, and so must be the order in which they are arranged as we show in Section II-A. For example, the filter channels during the forward pass are different from those of the "reconstructed" filters during the backward pass. The "reconstructed" filters during the backward pass are formed by taking the weights from the same channel across all filters, stacking those along the channel dimension and then spatially rotating the filter by 180 degrees. Similarly, the gradients need to be bundled together differently for the second convolution than the third. These two convolutions

are calculated per layer during the backward pass where we would like to avoid having to spill the gradients off-chip. Thus, there is no single way to pack them in memory (effectively pre-scheduling them) that would work for all cases where they are used. 3) Activations can be discarded after each layer during inference which is not the case during training where they are saved to be used by the backward pass. 4) Inference accelerators use narrow fixed-point arithmetic (e.g., 8b) whereas training today is done predominantly using floating-point, e.g., FP32, FP16, or Bfloat16. Floating-point values are typically wider making crossbars more expensive, and performing shift-and-add operations to extract bit-level sparsity is non-trivial for floating point. 5) Training starts with randomly initialized values that keep evolving throughout the training process.

To take advantage of a dynamic sparsity pattern that may appear in any or all the three tensors, we chose to target a solution that can *dynamically* eliminate ineffectual value pairs at runtime from all the three convolutions, or operations, in DNN training. For this solution to be energy efficient we would like to avoid the high hardware cost of an exhaustive interconnect that enables arbitrary moves of the operands as well as the energy and the latency cost of rearranging tensors in different ways to suite the different operations.

A. TensorDash

This section explains how *TensorDash* removes ineffectual value pairs when processing the example input tensors of Figure 3 and presents the hardware architecture of the *TensorDash* extension. This discussion presents *TensorDash* as an extension of *Tensorcore*-like processing units that perform 4 MACs per cycle. *TensorDash* can be used to extend any data-parallel unit. Section IV considers units of width up to 16 MAC and shows that the relative cost vs. performance benefits with *TensorDash* improve for wider units. Let us assume that we are processing the 3D convolution of two input tensors A and B and that our processing elements perform 4 MAC operations concurrently just like in *Tensorcore* units.

Figure 4 shows that *TensorDash* extends the TC with the following components: a) Staging buffers for A and B where the depth of each staging buffer is a design time parameter. For clarity, let us assume that this depth is 2, that is each staging buffer can hold up to two rows of values. Writes to these staging buffers are row-wide. There are four single-value-wide read ports each feeding directly to a multiplier input. As Fig. 3c shows, the connectivity per read port is sparse: each port can read one out of a limited set of values, four in our example, within the staging buffer. The set of values that each port can read out is different, but can overlap. A staging buffer is implemented as an array of registers with as many row-wide write ports as the number of rows it can hold. Each read port is implemented as a multiplexer with a sparse connectivity to a subset of the buffer registers. b) There is a *hardware scheduler* that accepts a bit vector from each staging buffer identifying which values are non-zero. For 2-deep staging buffers, the bit vectors would be 8b wide for our example. Each cycle the scheduler selects up to 4 effectual pairs from the staging buffers. It generates the control signals for the read ports, 2b per port for our example, so that the corresponding values are read out. The same control signal is shared among the corresponding ports in the two staging buffers, i.e., the same control signal goes to port p in the horizontal and vertical staging buffers so that operands from both move in tandem (4x2b control signals in total).

Figure 3c illustrates how *TensorDash* can effectively exploit sparsity even though it allows only a limited set of value movements per lane. There are two types of movement: in *time* only or *lookahead*, and in *space-time* or *lookaside*. The figure shows the set of possible movements for the third multiplier lane: it can either process the original dense value a_0^2, the next value in the same lane a_1^2 (lookahead), or it can *steal* a value from a step ahead in time from one of its two neighboring lanes a_1^1 or a_1^3 (lookaside). The movements possible by the other read ports are structurally identical relatively to their lanes and the ports are treated as if they are arranged into a ring with port 0 being adjacent to port 3. Each port can access a different set of values, however, these sets may overlap. Figures 3d and 3e show how *TensorDash* reduces processing time to the minimum two cycles using just a four-input multiplexer per multiplier input.

To improve performance, the staging buffers will need to be kept full as much as possible. For peak capability, an N-deep staging buffer needs N row-wide write ports, one per row (to enable filling it at once whenever it is drained in one cycle). The A and B scratchpads will have to be banked accordingly to sustain the higher read throughput needed when multiple rows are drained out of the staging buffers at once. For our example, dual-banked scratchpads are sufficient. In general, having as many banks as *lookahead* is more than enough and we found empirically that a lookahead of three is more than sufficient for TC-like units. For wider units with more MACs, even 2 lookahead is enough. We describe our PE configuration and the hardware scheduler next.

The Hardware Scheduler: Each PE in a *Tensorcore* accepts four pairs (A, B) of FP16 values and performs four MACs per cycle. In our preferred configuration, *TensorDash* adds a 4-deep staging buffer on each input side. As Fig. 5 shows, the staging buffer can hold four rows, each is 4-value wide, corresponding to the dense schedule for the current step (step +0) and the next three in time (+1, +2 and +3). For every lane there is a multiplexer which implements a sparse connectivity pattern. The figure shows the connections for lane 1. Besides the original "dense" schedule value, there are three lookahead and four lookaside options per input. For example, the multiplier for lane #1 can be given the value at lane 1 from the current time slot or up to 3 ahead. Alternatively, it can "steal" the values from neighboring lanes. For example, it can get the value from lane 2 that is one time step ahead or the value from lane 3 that is two steps ahead. Each lane has the same connectivity pattern which is shifted relative to its position (wrapping around the side edges). Each staging buffer also generates a 4x4b bit vector (using per value comparators), denoted as Z_A and Z_B for the A and B staging buffers respectively, indicating which

of their values are zero.

The scheduler accepts the two bit vectors Z_A and Z_B from the A and B staging buffers and generates two sets of signals. The first set is four MS_i, i=0...3 3b signals, one per input lane. These are used as the select signals for the per lane multiplexers. There is one MS_i signal per multiplier, and it is used by the multiplexers on both the A and B sides of that lane. The scheduler also produces a 2b AS signal that indicates how many rows of the staging buffer it has been able to drain, so that they can be replenished from the scratchpads which are banked to keep the buffers full.

The rest of this section describes the scheduler block. The Z_A and Z_B 4x4b bit vectors are first bit-wise ORed to produce a 4x4b bit vector Z. It indicates which pairs of (A,B) values have at least one zero. These pairs are ineffectual and can be skipped. The scheduler's goal is to select a movement per lane, for a total of 4 movements (MS_i signals) so that it processes as many of the remaining effectual (A,B) pairs as possible in one step. We will refer to the selection of movements that the scheduler makes for one step as a *schedule*.

For each lane i the scheduler uses a simple static priority scheme: among the eight options select the first available in the following order (notation is (step,lane) refer to Fig. 5): (+0,i) (dense schedule), (+1,i) lookahead 1 step, (+2,i) lookahead 2 steps, (+3,i) lookahead 3 steps, and then the lookaside options: (+1,i+1), (+1,i-1), (+2,i+2) and (+3,i+3). The scheduler uses an 8b-to-3b priority encoder per lane. Each priority encoder operates on an 8b subset of Z, corresponding to the candidate options for its lane, picks the first effectual option according to the aforementioned priority scheme, and produces the 3b MS_i signal. However, having all lanes make their selections independently may yield an invalid schedule; the same pair may be chosen by multiple lanes and end up being multiplied and accumulated more than once.

To ensure that the scheduler always produces a valid schedule, one where each value pair is selected *once*, we use a hierarchical scheme where scheduling is done in 4 levels as shown in Fig. 6. In each level, exactly one lane makes its decision independently using the *current* value of the Z vector as input. Given the promotion connectivity in Fig. 5, one lane per level is necessary to guarantee *by design* that lanes will not make overlapping choices. After a lane makes it selection it "removes" this options (OR gate) from the Z vector before passing it to the next level. Generating the AS signal is straightforward given the bits that are left in Z at the end. While we have described the above process in steps, the scheduler is *combinatorial* and operates in a single cycle.

Composing TensorDash Cores: So far we have described a single *TensorDash* processing element (PE) which can exploit sparsity on both operands. A *Tensorcore* can be implemented as a 4×4 tile of such PEs. While a PE can exploit reuse only temporally, spatial data reuse is also possible by having the PEs along the same row share the same B input and PEs along the same column share the same A input. For example, during the forward pass and for a convolutional layer, each row can be processing a different filter, whereas columns can be processing different windows. In this arrangement, each PE would be processing a unique combination of B and A inputs. Skipping zeros on both A and B sides remains possible if we use per PE schedulers and staging buffers.

We opt for extracting sparsity from only the B side since there is sufficient sparsity in one of the operands in each of the three major operations to extract significant benefits. Figure 7 shows a simplified core with 2×2 tile configuration. Each row of PEs uses a common scheduler and shares the same staging buffer and multiplexer block on the B side. For the A side, we use a single staging buffer per column and a dedicated multiplexer block per PE. The A-side multiplexer blocks per row share the same MS_i signal from the row scheduler. Each scheduler now need to see only the Z vector from the corresponding B-side staging buffer. Under this tiling configuration, synchronization between rows of PEs has to be enforced where all rows have to wait for the row with the longest B-side schedule. This is to ensure all rows advance in tandem to the next set of values on the A-side. For example, assuming 3-deep staging buffers and 2×2 tile configuration, if row 0 scheduled the 3 time steps of its staging buffer in 2 processing cycles but row 1 did so in 1 cycle, row 1 has to stall waiting for row 0 before they both advance to the next set of values on the A-side. We study the effect of synchronization in Section IV-D. The designs we evaluate for both *TensorDash* and the baseline use *Tensorcore* (4×4 tile of PEs) as the main building block.

Tensor Layout and Transposing: During training, each tensor is used in two the three major computations. For example, the weights in the forward pass are convolved with the activations whereas in the backward pass they are convolved with the output gradients. In each operation the group of weights that contribute to an output value is different. This is true for the weights, activations and gradients. This has implications for the memory hierarchy which needs to supply the data in an appropriate order to the PEs. When a tensor is used in only one way it is possible to statically layout its values in memory so that they can be easily served using wide accesses off- and on-chip. However, during training the layout that serves well one of the computations will not be able to serve well the other. Fortunately, it is possible to arrange values in memory so that they can be easily fetched for all use cases. The key is the ability to transpose tensors as needed. For this purpose, we use a tensor layout where values are stored in groups of 4x4 values. The group is formed by taking four blocks of values adjacent along the X dimension. Each of these blocks contains four consecutive values along the channel dimension. The starting coordinates for each 4x4 value group are aligned by four along the X and the channel dimensions. Finally, the groups constituting a tensor are allocated in memory space in channel, Y, X order.

When fetching values from off-chip, each group can be written directly to the multi-bank on-chip memories so that each 4-value block is copied directly to a bank. As a result, the PE can now directly access any block of 4 values consecutive along the channel dimension in a single step. When transposing is needed, we use on-chip transposers between the on-chip

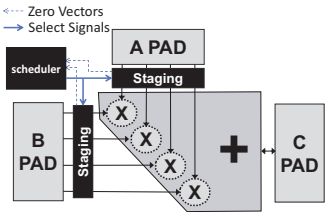


Fig. 4: *TensorDash* Processing Element.

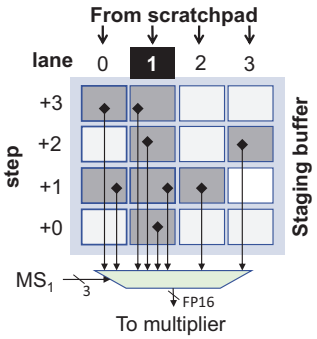


Fig. 5: Staging buffer connectivity for lane #1.

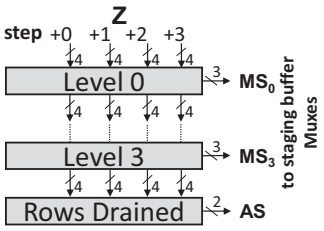


Fig. 6: *TensorDash*'s Scheduler.

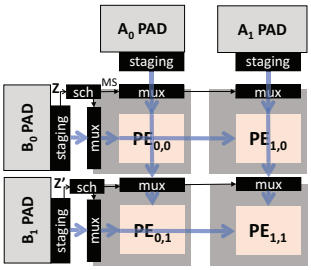


Fig. 7: 2x2 *TensorDash* Tile.

memory banks and the tile scratchpads. The number of transposers is chosen so that the memory system can keep the tiles busy. Each transposer reads four 4-value blocks from their banks using 4-value wide accesses and copies those into its internal 4x4 buffer. The transposer then can provide four blocks of four values each composed of a single value from each of the four original blocks read from memory effectively transposing the tensor. For example, it can supply a block comprising all values that appear first within their original block, or all that appeared third. This transpose scheme is sufficient to serve well all the three computation operations involved during training.

IV. Evaluation

DNN models: As shown in Table I, we evaluate *TensorDash* on state-of-the-art DNN models covering a wide range of applications: 1) image classification trained on ImageNet [63]: SqueezeNet [37], VGG [65] and ResNet-50 [32], 2) scene understanding: img2txt [70] trained on Microsoft COCO dataset [47], 3) natural language modeling: including BERT [20] the Transformer-based model from Google trained on the GLUE dataset [71] and SNLI which is trained on the Stanford Natural Language Inference corpus [7], 4) object detection and segmentation: Facebook's Detectron2 model trained on Microsoft COCO dataset [47], and 5) recommendation system: Neural Collaborative Filtering (*NCF*) [33] trained on the MovieLens 20M movie ratings dataset [1]. To show how *TensorDash* benefits from techniques that incorporate pruning during training of dense models, we trained two variants of ResNet-50 following: 1) the dynamic sparse re-parameterization technique of Hesham *et al.* [55], and 2) the sparse momentum technique of Dettmers *et al.* [19]. For both techniques we target 90% sparsity. We show results only for the latter technique due to space limitations.

Collecting Traces: We trained all models on an RTX 2080 Ti GPU using the PyTorch implementations as released by the original authors. We trained each model for as many epochs as needed for it to converge to its best reported output accuracy. For each epoch, we sampled one randomly selected batch and traced the operands of the three operations shown in Eqs. (1) to (3). The batch size is different per model due to their different GPU memory requirements. It ranges from as low as 64 and up to 143 samples per batch.

***TensorDash* and *Tensorcore*-based Baseline**			
# of TCs	256	AM SRAM	128KB×64 Banks
TC core	4 × 4 PEs	BM SRAM	128KB×64 Banks
PE MACs/Cycle	4	CM SRAM	128KB×64 Banks
Staging Buff Depth	4	Scratchpads	256KB total
Tech Node	65nm	Frequency	500 MHz
Off-Chip Memory		16GB 4-channel LPDDR4-3200	
Peak FLOPS		16.4 TFLOPS	

TABLE III: *Tensorcore* baseline and *TensorDash*-enhanced configurations.

Accelerator Modeling: We developed a custom cycle-accurate simulator to model performance. We have performed extensive microbenchmark stress tests to validate the fidelity of the simulator and we plan to release it. Table III lists the default configurations for the *Tensorcore*-based baseline including the *TensorDash* extension. To model area and power consumption, all designs were implemented in Verilog, synthesized via the Synopsys Design Compiler and layout was produced via Cadence Innovus and for a 65nm TSMC technology which is the best that is available to us due to licensing restrictions. Power was estimated by capturing circuit activity via Mentor Graphics' ModelSim which was then passed on to Innovus. CACTI [36] was used to model the area and energy consumption of the on-chip shared SRAM memories which are divided into three heavily banked chunks: AM, BM, and CM. We used CACTI also to model the area and energy consumption of the SRAM scratchpads (SPs). Finally, energy and latency for off-chip accesses were modelled via Micron's DRAM model [53]. Both the baseline and *TensorDash* architectures compress zero values off-chip using the CompressingDMA method [62].

A. Performance

Fig. 8 reports the speedup achieved by incorporating *TensorDash* in the *Tensorcore* architecture for each model and for each of the three operations: (i) $A \star W$, (ii) $A \star G$, and (iii) $W \star G$. Since the amount of sparsity and its pattern in each of the tensors differ across models, layers, and training phase, the speedup varies. On average, *TensorDash* accelerates execution by 1.95× and never introduces any slowdown.

TensorDash benefits all models. The benefits for BERT and Dtctron2 come from their backward pass only. Dtcron2

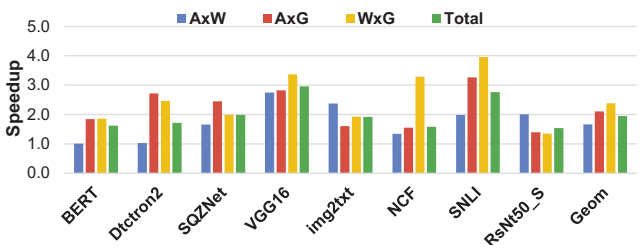


Fig. 8: Speedup of *TensorDash* over the baseline architecture.

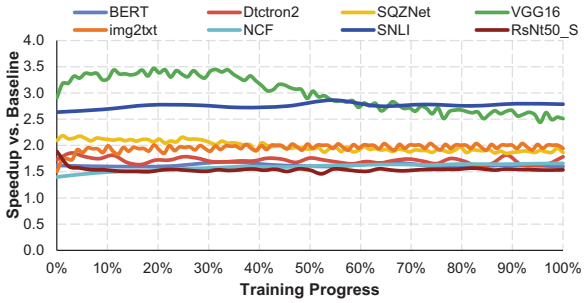


Fig. 9: Speedup with *TensorDash* as training progresses.

consists of:1) a pretrained ResNet-50 backbone, 2) a feature pyramid network (*FPN*) [46] that extracts high-level semantic feature maps at different scales, 3) a region proposal network (*RPN*) [31] that proposes candidate object bounding boxes, 4) a network head performing bounding-box recognition, classification, and regression, and 5) a network head for mask prediction that is applied separately to each region of interest. The two front-end convolution layers of the FPN and RPN, which dominate the execution time, show negligible input activation sparsity. However, the use of ReLU directly after these dominating layers results in significant (73%-94%) sparsity in the gradients for these layers that *TensorDash* exploits during the backward pass. BERT is dominated by fully-connected-like layers which exhibit little sparsity in their weights and activations. But, despite not using ReLU, we see sparsity in its gradients during the backward pass where the attention mechanism of its encoder and decoder layers results in roughly 60% sparsity in the gradients for most of the layers.

ResNet-50, VGG16, and SQZNet are among the models that use ReLU and thus benefit from the sparsity it generates. In ResNet-50 the benefits are lower during the backward pass. This is predominantly caused by the use of batch normalization (*BatchNorm*) layers between each convolutional layer and the subsequent ReLU layer. A BatchNorm layer absorbs almost all the sparsity in the gradients. Fortunately, however, there is still sparsity in either the activations or the weights which *TensorDash* exploits. The use of in-training pruning creates considerable sparsity in the weights, especially for the smaller back-end layers, which *TensorDash* capitalizes on during the $W \star G$ operation.

SNLI performs natural language inference task through recognizing textual entailment between pairs of human-written English sentences. It includes two fully connected (*FC*) projections layers, two LSTM encoders, and four fully connected classifier layers. We observe significant sparsity in the gradients and activations which explains the observed benefits with *TensorDash*. The gradients exhibit more than 95% sparsity due to the nature of the task and the use of ReLU activations. Input activation sparsity is 63% and 60% for the two front-end projection layers, 0% for the first classification layer, and over 94% for other layers.

The NCF recommendation system consists of four FC layers with the first layer being the largest. The gradients are 83% sparse which benefits the $W \star G$ operation. However, this operation is not performed for the first layer which is the most expensive and that also exhibits no activations sparsity.

B. Speedup Over Time

Fig. 9 shows execution time speedup with *TensorDash* during training from the first epoch up until convergence. Generally, improvements with *TensorDash* are fairly stable throughout the entire training process which suggests that the underlying phenomena that *TensorDash* exploits are neither transient nor caused by initialization.

The measurements reveal two trends. For ResNet50 which uses a in-training pruning method, speedups are slightly higher during the first few epochs, and then reduce and stabilize at around 1.5×. Similar, albeit slightly more subdued behavior is seen for the other dynamic sparse re-parameterization technique [55]. This behavior is due to the pruning algorithm which starts by aggressively pruning many weights at the beginning which the training process then "reclaims" to recover the accuracy of the model.

For the dense image classification models, where most of the sparsity that *TensorDash* exploits originates from the activations and the gradients, the speedup tends to follow an overturned U-shape curve. This behavior is more pronounced for VGG16 model where the benefits are initially lower due to the random initialization of the model. Then benefits rapidly increase during the first few epochs as the model is quickly improving by learning what features of the input data are irrelevant for the task. This translates to rapid increases in sparsity in both the activations and the gradients. The speedup then stabilizes until $40\% - 50\%$ of the training process is reached. It then gradually decreases as we enter the second half of the training process where the model starts to extract some of the less-important previously discarded features to improve accuracy. During the final quarter of the training process, the speedup stabilizes as the model parameters are very close to their final values and thus the sparsity of the activations and gradients stabilizes. Rhu *et al.* have made similar observations when studying sparsity during training for the purpose of compressing data off-chip [62].

C. Area Overhead and Energy Efficiency

Table IV reports a breakdown of the area and power. Even without taking the on- and off-chip memories into account, the area and power overhead of *TensorDash* is small; only

TABLE IV: Area and power consumption breakdown of *TensorDash* vs. baseline TC. On-chip memory and scratchpads are not included.

	Area (mm^2)		Power (mW)	
	TensorDash	Baseline	*TensorDash*	Baseline
Compute Cores	68.74		23,748	
Transposers	0.37		44.4	
Schedulers+B-Side MUXes	1.37	-	187.9	-
A-Side MUXes	3.63	-	283.8	-
Staging Buffers	4.91	-	1,638.4	-
Total	**79.01**	**69.11**	26,144	23,793
Normalized	**1.14×**	**1×**	**1.09×**	**1×**
	Overall Energy Efficiency		**1.5×**	**1×**

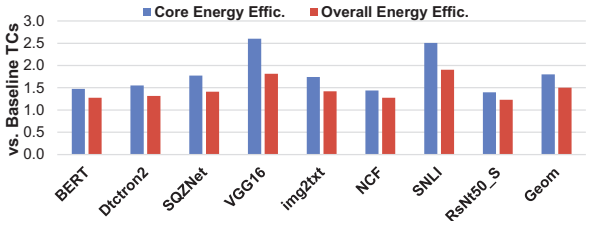


Fig. 10: Energy efficiency of *TensorDash* over the baseline.

14% extra silicon area and 9% more power are needed for the schedulers, staging buffers, and the back-end shuffling MUXs. Given the speedup with *TensorDash*, the compute logic alone becomes on average 1.8× more energy efficient compared to the plain *Tensorcore*. Fig. 10 reports per model energy efficiency for the compute logic and the whole chip.

Each of the on-chip AM, BM and CM memory chunks require 58.6 mm^2 of area whereas the scratchpads require a total of 3.95 mm^2 for the baseline and 5.9 mm^2 for *TensorDash* due to more banking. In total, when considering both the compute and memory area of the whole chip, the area overhead of *TensorDash* stands at only 4.8%. As Fig. 10 shows, when we take the accesses to the on-chip memories, the scratchpads, and the off-chip DRAM into account, introducing *TensorDash* improves the overall energy efficiency of the *Tensorcore* architecture by 1.5×.

Fig. 11 reports the energy consumed with *TensorDash* relative to the baseline. The measurements also show a breakdown of the energy consumed across the three main components: the off-chip data transfers, core logic, and the on-chip memory modules. *TensorDash* significantly reduces the energy consumption of the compute cores which dominates the energy consumption of the system.

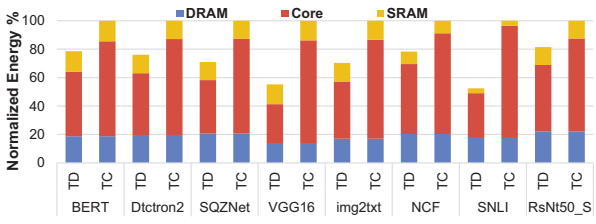


Fig. 11: Energy consumption breakdown of *TensorDash* normalized to the Baseline: off-chip DRAM, compute logic and on-chip SRAM.

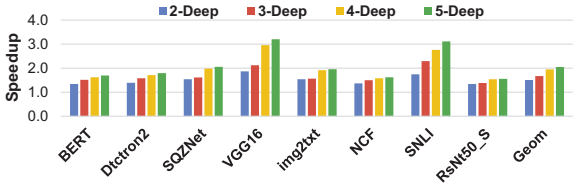


Fig. 12: *TensorDash* speedup with staging buffer depth.

D. Analysis

• **Staging Buffer Depth/Lookahead:** The depth of the staging buffers dictates the theoretical peak speedup since it limits the maximum number of time steps that can be skipped at once. For an N-deep buffer the maximum speedup is $N\times$. To study the effect of depth configurations on performance, we sweep depths from 2 up to 5. The 2-deep and 3-deep configurations implement four possible *promotions* per lane instead of eight for a more balanced implementation cost vs. performance. Fig. 12 shows that the average speedup grows from 1.5× (2-deep) to 2.05× (5-deep). The returns tapering off from 4-deep to 5-deep does not justifying the extra cost.

The staging buffer depth can be decided by taking two factors into account: 1) hardware overhead: deeper buffers are more expensive, demand more scratchpad banks and wider MUXes per multiplier lane to reasonably cover the additional candidate values. 2) PE lane width: that is, the number of multiplier lanes per PE. On one side, for wider baseline PEs we expect deeper staging to be less effective due to the lower probability to empty multiple time steps for all lanes. On the other side, this is balanced by better-spread, less overlapping, and higher pattern coverage across the lanes with wider PEs. This improves the chances of more effective promotions and, thus, to skip more steps. This trade-off is illustrated in the following *Tile Geometry: Wider PE* study.

• **Tile Geometry:** We study the performance behavior of the *TensorDash* PE when it is used to compose tiles. For this purpose, we vary the number of PE rows and columns per tile and study how this affects performance. As the tile geometry scales, stalls may occur due to inter-PE synchronization which in turn is caused by work imbalance.

Rows: Fig. 13 shows how the performance of *TensorDash* changes as the number of rows per tile varies from 1 and up to 16 while the number of columns is fixed at 4. The average speedup decreases from 2.2× for a tile with 1 row to 1.8× when the tile has 16 rows. Since all PEs have to wait for the slowest one, the more rows the more frequent stalls due to work imbalance will occur. As we scale up the number of rows per tile, value density imbalance across rows increases leading to more stalls where *all* rows have to wait for the row with the densest value stream. The main reason why this occurs is that the non-zero activations and gradients tend to cluster in certain 2D feature maps whereas the other 2D maps become more sparse. This clustering phenomenon is fundamental in such models especially towards the deeper layers where each filter is trained to extract specific high level features. However, as the results show, *TensorDash* tackles row imbalance reasonably well. Lookaside is effective in spreading work from what

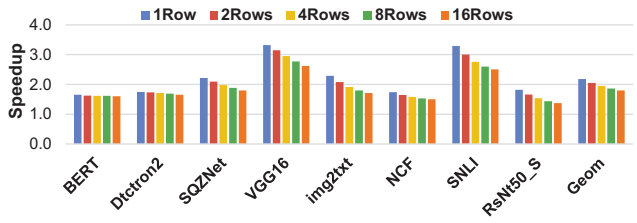


Fig. 13: *TensorDash* speedup vs. number of PE rows.

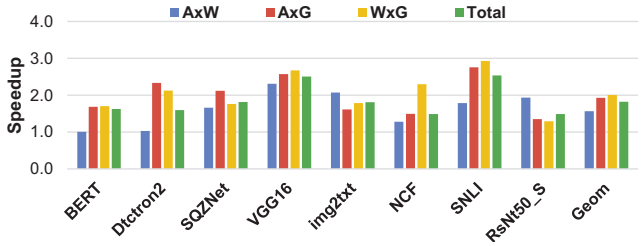


Fig. 14: Speedup of *TensorDash* with wider 16-MAC PEs and 3-deep staging buffers over a similarly configured baseline.

would otherwise be "struggler" lanes. This phenomenon is more pronounced for $A \times G$, the second backward convolution, where the 2D feature maps of the activations and the gradients are convolved.

Columns: We vary the number of columns per *Tensorcore* from 4 to 16 while keeping the number of rows at 4. Since we exploit sparsity only on the row side, increasing the number of columns does not affect performance as much compared to an equivalently scaled baseline. All rows still have to wait for the row with the densest stream. However, increasing the columns allows us to process more windows in parallel in convolutional layers while sharing the same schedule along the rows. We noticed a negligible drop in the speedup of some models that does not exceed 3% and is predominantly due to fragmentation caused by layer dimensions.

Wider Processing Elements: We experiment with wider *Tensorcore* configurations that have more multiplier lanes. We study a *TensorDash* design that still allowed only 8 promotions per lane. We found that a wider *Tensorcore* performs well even with shallower staging buffers and thus less hardware overhead. It achieves almost the same performance and energy efficiency as the narrow *Tensorcore* with 4-deep buffers. Fig. 14 shows that *TensorDash* with 3-deep staging buffers on top of 16-wide PEs improves performance on average by 1.85×. Meanwhile, the area overhead is reduced to 10% and 3.5% for the compute logic and the whole chip respectively. As a result of the sparse connectivity pattern used, lanes with non-overlapping connectivity patterns could be grouped for scheduling in one level which permits a scheduler with just 6 levels and that is not in the critical path.

• **Connectivity Pattern:** We have experimented with tens of connectivity patterns and found that as long as there are enough lookaside options, the specific patterns does not affect performance significantly (±5% relative performance differences). Intuitively, the pattern should be selected such that: 1) there is a balance between the number of lookahead and lookaside options given a specific MUX width, and 2) lookaside options are spread across the future steps of the neighboring lanes. The robustness of our chosen connectivity paterns is further demonstrated by the following experiment where we demonstrate *TensorDash*'s effectiveness with synthetically generated random tensors.

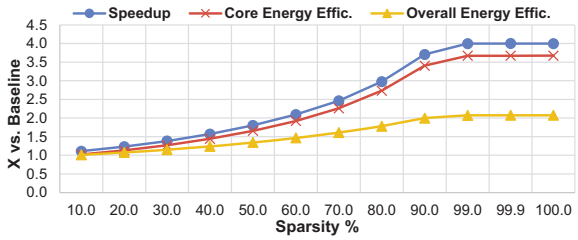


Fig. 15: *TensorDash* speedup for randomly sparse tensors.

• **Effect of Tensor Sparsity:** To determine whether *TensorDash* remains effective regardless of the sparsity structure, we experimented with synthetically generated sparse tensors with sparsity levels ranging from 10% up to 99.99%. We used the architecture of the third convolution layer from SQZNet but populated the tensors using randomly generated values. For each level of sparsity, we generated 10 tensor samples. We then performed all three operations for each sample using these generated tensors. We report the average across all samples for each sparsity level (the maximum deviation in measured speed was below 5%). Recall that given the 4-deep staging buffers we use, the maximum possible speedup with *TensorDash* even if the tensor contains only zeros is 4×. As Fig. 15 shows, performance and energy efficiency with *TensorDash* closely follows the amount of sparsity in the input. The figure shows that when the ideal speedup is below 4×, *TensorDash* comes close to what is ideally possible. For example, with 20% sparsity, an optimal machine would be 1.25× faster. *TensorDash* is approximately 1.23× faster. For 90% sparsity, an ideal machine would be 10× faster. The experiment shows that *TensorDash* comes close to its ideal 4× speedup being 3.7× faster and reaching to 3.99× for the 99% sparsity level. Core energy efficiency closely follows the speedup, however within a narrower envelop due to the 9% power overhead of *TensorDash* components. For the highest sparsity level we study 99.99%, the maximum possible core energy efficiency is saturating at *(max possible speedup/core power overhead)* $= 4/1.09 = 3.67$. Overall energy efficiency is predominantly a function of layer dimensions which dictate memory transfers to/from on-chip SRAM and off-chip DRAM. Overall energy efficiency follows a similar trend but saturates at 2.1×.

• **Training with Bfloat16:** Recent work showed that deep neural networks could be trained using other floating-point data types such as bfloat16 [26], [41], [73]. We implemented *TensorDash* and baseline configurations that use bfloat16 arithmetic. Even when we consider only the compute logic, our synthesis and layout results show that the area and power overheads of *TensorDash* remain low at 1.16× and 1.1× respectively. The various components scale differently as the

data type changes. For example, while hardware overhead of the scheduler and shufflers does not change when we go from FP16 to bfloat16, the multipliers shrink. When the on-chip memory structures are taken into account, the area overhead is 4.9%. In terms of energy efficiency, the compute logic with *TensorDash* is on average 1.76× more energy efficient than the baseline. When accesses to the on-chip and the off-chip memory are taken into account, introducing *TensorDash* boosts overall energy efficiency by 1.48×.

• **A Model with Virtually No Sparsity:** We experimented with GCN [15], a natural language processing model which we trained on the Wikitext-2 dataset [51]. It exhibits virtually no sparsity in the activations, gradients, and weights. Still, *TensorDash* improves performance by 1% since a few layers exhibit about 5% sparsity. *TensorDash* overall energy efficiency is 0.5% lower than the baseline.

V. Related Work

The architecture of choice for training has been the graphics processor. Neural networks and GPUs have evolved almost symbiotically during the last few years with GPUs introducing features to aid inference and training [8]. XeonPhi is another architecture that is well suited to this type of data-parallel workload [61]. Scaledeep is a scalable architecture for training that utilizes heterogeneous tiles and chips, an optimized network topology, low-overhead hardware-assisted synchronization, and optimized model partitioning [69]. DaDianNao, one of the earliest accelerator architectures targeting primarily inference whose tiles, however, could be fused to support 32b arithmetic for training [12]. Newer versions of the TPU also support training [39]. Plasticine does not target machine learning exclusively but a wide set of parallel computation patterns which include those needed for stochastic gradient descent [59]. Caterpillar provides hierarchical support for collective communication semantics to provides the flexibility needed to efficiently train various networks with both stochastic and batched gradient descent-based techniques [44]. NXT is a near-memory accelerator comprising several general purpose cores and specialized co-processors targeting both inference and training [64]. Intel's NNP-T (Spring Crest) supports both FP32 and FP16 [76]. Cerebras' Wafer Scale Engine includes hundreds of thousands of Sparse Linear Algebra (SLA) Cores able to filter computation with zeros. In their dataflow architecture, computation is only triggered when a non-zero data is received by the SLA core [9].

Bit-Tactical uses a software scheduler plus a sparse interconnect to exploit the static sparsity in the weights of pruned models during inference [18]. For training, however, the dynamic nature of sparsity makes this approach impractical. The overhead of invoking a software scheduler per layer/sample/convolution is prohibitive in terms of latency and energy. Moreover, bundling the weights in a specific order, pre-scheduling these bundles and packing them in memory is possible for inference since the weights are being used only in the forward pass where the weights and activations are accessed in one specific order. Unfortunately, during training each tensor is accessed in two different orders across the three convolutions.

TensorDash is a plug-and-play element that exploits dynamic sparsity and can be used to compose processing tiles. As such it is not meant as a competitor for the overall accelerator architecture. That said, in every case there will be several considerations that need close attention and evaluation.

Sparse tensor operations appear in many application domains and specialized hardware designs have been receiving constant attention as exemplified by recent proposals [34], [57]. Compared to other application domains, neural network training in general encounters tensors of comparatively low sparsity whose pattern is not known in advance.

VI. Conclusion

TensorDash is a low-level processing element for building accelerators for the datacenter and the "edge". There is an ever increasing body of work for accelerating training in software, hardware or both. While *TensorDash* will interact with several of these methods, it is at first-order complementary with many since it operates at the very low-level of the MAC units.

While here we studied TensorDash for training acceleration, it can also be used for inference acceleration. In addition, while in this work we did not pre-schedule the input tensors in memory, it is possible to do further reducing memory footprint, traffic, and energy during training and inference [49]. Pre-scheduled tensors can first be expanded from the pre-scheduled form to their corresponding "dense" form in the staging buffers. This is implemented using a sparse interconnect that mirrors the interconnect *TensorDash* used in this work to select the effectual operands. For weights the pre-scheduling can be done in advance and in software. For activations, it can be done on the output of the preceding layer. Another hardware scheduler, identical to the one described in this work pre-schedules the activations as they are produced at the output of a layer.

Acknowledgement

This work was supported by the NSERC COHESA Strategic Research Network and an NSERC Discovery Grant. The University of Toronto maintains all rights on the technologies described.

References

[1] "Movielens 20m dataset," https://grouplens.org/datasets/movielens/20m/.

[2] "NVIDIA Tesla V100 GPU Achitecture," 2017. [Online]. Available: https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf

[3] "Cerebras CS1," 2019. [Online]. Available: https://www.cerebras.net/product/

[4] "Gaudi training platform white paper," 2019. [Online]. Available: https://habana.ai/wp-content/uploads/2019/06/Habana-Gaudi-Training-Platform-whitepaper.pdf

[5] J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. Enright Jerger, and A. Moshovos, "Cnvlutin: Ineffectual-Neuron-Free Deep Neural Network Computing," in *Intl' Symp. on Computer Architecture*, 2016.

[6] D. Amodei, D. Hernadez, G. Sastry, J. Clark, G. Brockman, and I. Sutskever, "Open AI Blog." [Online]. Available: https://openai.com/blog/ai-and-compute/

[7] S. R. Bowman, G. Angeli, C. Potts, and C. D. Manning, "A large annotated corpus for learning natural language inference," in *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*. Lisbon, Portugal: Association for Computational Linguistics, Sep. 2015, pp. 632–642. [Online]. Available: https://www.aclweb.org/anthology/D15-1075

[8] J. Burgess, "RTX ON - the NVIDIA TURING GPU," in *2019 IEEE Hot Chips 31 Symposium (HCS), Cupertino, CA, USA, August 18-20, 2019*, 2019, pp. 1–27. [Online]. Available: https://doi.org/10.1109/HOTCHIPS.2019.8875651

[9] Cerebras Systems, "Cerebras Wafer Scale Engine: An Introduction," "https://www.cerebras.net/wp-content/uploads/2019/08/Cerebras-Wafer-Scale-Engine-An-Introduction.pdf", 2019.

[10] Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks," in *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3. IEEE Press, 2016, pp. 367–379.

[11] Y.-H. Chen, T. Krishna, J. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," *IEEE Journal of Solid-State Circuits*, vol. 52, no. 1, Jan 2017.

[12] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, and O. Temam, "DaDianNao: A Machine-Learning Supercomputer," in *Intl' Symp. on Microarchitecture*, 2014.

[13] J. Choi, Z. Wang, S. Venkataramani, P. I.-J. Chuang, V. Srinivasan, and K. Gopalakrishnan, "PACT: Parameterized clipping activation for quantized neural networks," *arXiv preprint arXiv:1805.06085*, 2018.

[14] 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," in *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*, 2018. [Online]. Available: https://openreview.net/forum?id=H135uzZ0-

[15] Y. N. Dauphin, A. Fan, M. Auli, and D. Grangier, "Language modeling with gated convolutional networks," in *Proceedings of the 34th International Conference on Machine Learning - Volume 70*, ser. ICML'17. JMLR.org, 2017, pp. 933–941. [Online]. Available: http://dl.acm.org/citation.cfm?id=3305381.3305478

[16] C. De Sa, M. Leszczynski, J. Zhang, A. Marzoev, C. R. Aberger, K. Olukotun, and C. Ré, "High-accuracy low-precision training," *arXiv preprint arXiv:1803.03383*, 2018.

[17] J. Dean, G. S. Corrado, R. Monga, K. Chen, M. Devin, Q. V. Le, M. Z. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang, and A. Y. Ng, "Large scale distributed deep networks," in *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, ser. NIPS'12. USA: Curran Associates Inc., 2012, pp. 1223–1231. [Online]. Available: http://dl.acm.org/citation.cfm?id=2999134.2999271

[18] A. Delmas Lascorz, P. Judd, D. M. Stuart, Z. Poulos, M. Mahmoud, S. Sharify, M. Nikolic, K. Siu, and A. Moshovos, "Bit-tactical: A software/hardware approach to exploiting value and bit sparsity in neural networks," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '19. New York, NY, USA: ACM, 2019, pp. 749–763. [Online]. Available: http://doi.acm.org/10.1145/3297858.3304041

[19] T. Dettmers and L. Zettlemoyer, "Sparse networks from scratch: Faster training without losing performance," *arXiv preprint arXiv:1907.04840*, 2019.

[20] J. Devlin, M. Chang, K. Lee, and K. Toutanova, "BERT: pre-training of deep bidirectional transformers for language understanding," *CoRR*, vol. abs/1810.04805, 2018. [Online]. Available: http://arxiv.org/abs/1810.04805

[21] M. Drumond, T. Lin, M. Jaggi, and B. Falsafi, "Training DNNs with hybrid block floating point," in *Proceedings of the 32Nd International Conference on Neural Information Processing Systems*, ser. NIPS'18. USA: Curran Associates Inc., 2018, pp. 451–461. [Online]. Available: http://dl.acm.org/citation.cfm?id=3326943.3326985

[22] T. Elsken, J. H. Metzen, and F. Hutter, "Neural architecture search: A survey," *Arxiv preprint arxiv:1808.05377*, 2018.

[23] R. D. Evans, L. F. Liu, and T. Aamodt, "JPEG-ACT: A frequency-domain lossy DMA engine for training convolutional neural networks," in *Proceedings of the 47th Annual International Symposium on Computer Architecture*, ser. ISCA '20. ACM, 2020.

[24] M. Golub, G. Lemieux, and M. Lis, "Dropback: Continuous pruning during training," *arXiv preprint arXiv:1806.06949*, 2018.

[25] A. Gondimalla, N. Chesnut, M. Thottethodi, and T. N. Vijaykumar, "SparTen: A sparse tensor accelerator for convolutional neural networks," in *Proceedings of the 52Nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '52. New York, NY, USA: ACM, 2019, pp. 151–165. [Online]. Available: http://doi.acm.org/10.1145/3352460.3358291

[26] Google, "Using bfloat16 with tensorflow models," https://cloud.google.com/tpu/docs/bfloat16.

[27] U. Gupta, B. Reagen, L. Pentecost, M. Donato, T. Tambe, A. M. Rush, G. Wei, and D. Brooks, "MASR: A modular accelerator for sparse rnns," in *28th International Conference on Parallel Architectures and Compilation Techniques, PACT 2019, Seattle, WA, USA, September 23-26, 2019*. IEEE, 2019, pp. 1–14. [Online]. Available: https://doi.org/10.1109/PACT.2019.00009

[28] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding," in *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, Y. Bengio and Y. LeCun, Eds., 2016. [Online]. Available: http://arxiv.org/abs/1510.00149

[29] S. Han, J. Pool, J. Tran, and W. Dally, "Learning both weights and connections for efficient neural network," in *Advances in neural information processing systems*, 2015, pp. 1135–1143.

[30] B. Hassibi and D. G. Stork, "Second order derivatives for network pruning: Optimal brain surgeon," in *Advances in Neural Information Processing Systems 5*, S. J. Hanson, J. D. Cowan, and C. L. Giles, Eds. Morgan-Kaufmann, 1993, pp. 164–171. [Online]. Available: http://papers.nips.cc/paper/647-second-order-derivatives-for-network-pruning-optimal-brain-surgeon.pdf

[31] K. He, G. Gkioxari, P. Dollar, and R. Girshick, "Mask R-CNN," in *The IEEE International Conference on Computer Vision (ICCV)*, Oct 2017.

[32] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *CoRR*, vol. abs/1512.03385, 2015. [Online]. Available: http://arxiv.org/abs/1512.03385

[33] X. He, L. Liao, H. Zhang, L. Nie, X. Hu, and T. Chua, "Neural collaborative filtering," *CoRR*, vol. abs/1708.05031, 2017. [Online]. Available: http://arxiv.org/abs/1708.05031

[34] K. Hegde, H. A. Moghaddam, M. Pellauer, N. C. Crago, A. Jaleel, E. Solomonik, J. S. Emer, and C. W. Fletcher, "Extensor: An accelerator for sparse tensor algebra," in *Proceedings of the 52nd Annual IEEE / ACM International Symposium on Microarchitecture, MICRO 2019, Columbus, OH, USA, October 12-16, 2019*. ACM, 2019, pp. 319–333. [Online]. Available: https://doi.org/10.1145/3352460.3358275

[35] K. Hegde, J. Yu, R. Agrawal, M. Yan, M. Pellauer, and C. W. Fletcher, "UCNN: Exploiting computational reuse in deep neural networks via weight repetition," in *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ser. ISCA '18. Piscataway, NJ, USA: IEEE Press, 2018, pp. 674–687. [Online]. Available: https://doi.org/10.1109/ISCA.2018.00062

[36] HewlettPackard, "CACTI," https://github.com/HewlettPackard/cacti.

[37] 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. [Online]. Available: http://arxiv.org/abs/1602.07360

[38] A. Jain, A. Phanishayee, J. Mars, L. Tang, and G. Pekhimenko, "Gist: Efficient data encoding for deep neural network training," in *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ser. ISCA '18. Piscataway, NJ, USA: IEEE Press, 2018, pp. 776–789. [Online]. Available: https://doi.org/10.1109/ISCA.2018.00070

[39] 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*, ser. ISCA '17. New York, NY, USA: ACM, 2017, pp. 1–12. [Online]. Available: http://doi.acm.org/10.1145/3079856.3080246

[40] P. Judd, A. D. Lascorz, S. Sharify, and A. Moshovos, "Cnvlutin2: Ineffectual-activation-and-weight-free deep neural network computing," *CoRR*, vol. abs/1705.00125, 2017. [Online]. Available: http://arxiv.org/abs/1705.00125

[41] D. D. Kalamkar, D. Mudigere, N. Mellempudi, D. Das, K. Banerjee, S. Avancha, D. T. Vooturi, N. Jammalamadaka, J. Huang, H. Yuen, J. Yang, J. Park, A. Heinecke, E. Georganas, S. Srinivasan, A. Kundu, M. Smelyanskiy, B. Kaul, and P. Dubey, "A study of BFLOAT16 for deep learning training," *CoRR*, vol. abs/1905.12322, 2019. [Online]. Available: http://arxiv.org/abs/1905.12322

[42] U. Köster, T. J. Webb, X. Wang, M. Nassar, A. K. Bansal, W. H. Constable, O. H. Elibol, S. Gray, 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 *Proceedings of the 31st International Conference on Neural Information Processing Systems*, ser. NIPS'17. USA: Curran Associates Inc., 2017, pp. 1740–1750. [Online]. Available: http://dl.acm.org/citation.cfm?id=3294771.3294937

[43] Y. LeCun, J. S. Denker, and S. A. Solla, "Optimal brain damage," in *Advances in Neural Information Processing Systems 2*, D. S. Touretzky, Ed. Morgan-Kaufmann, 1990, pp. 598–605. [Online]. Available: http://papers.nips.cc/paper/250-optimal-brain-damage.pdf

[44] Y. Li and A. Pedram, "CATERPILLAR: Coarse grain reconfigurable architecture for accelerating the training of deep neural networks," *2017 IEEE 28th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, Jul 2017. [Online]. Available: http://dx.doi.org/10.1109/ASAP.2017.7995252

[45] H. Liao, J. Tu, J. Xia, and X. Zhou, "DaVinci: A scalable architecture for neural network computing," in *2019 IEEE Hot Chips 31 Symposium (HCS)*, 2019, pp. 1–44.

[46] T. Lin, P. Dollár, R. B. Girshick, K. He, B. Hariharan, and S. J. Belongie, "Feature pyramid networks for object detection," *CoRR*, vol. abs/1612.03144, 2016. [Online]. Available: http://arxiv.org/abs/1612.03144

[47] 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. [Online]. Available: http://arxiv.org/abs/1405.0312

[48] S. Liu, Z. Du, J. Tao, D. Han, T. Luo, Y. Xie, Y. Chen, and T. Chen, "Cambricon: An instruction set architecture for neural networks," in *2016 IEEE/ACM Intl' Conf. on Computer Architecture (ISCA)*, 2016.

[49] M. Mahmoud, I. E. Vivancos, O. Awad, A. H. Zadeh, G. Pekhimenko, J. Albericio, and A. Moshovos, "Tensordash: Exploiting sparsity to accelerate deep neural network training and inference," *Arxiv preprint cs.AR arXiv:2009.00748*, Sep. 2020.

[50] R. Mayer and H. Jacobsen, "Scalable deep learning on distributed infrastructures: Challenges, techniques and tools," *CoRR*, vol. abs/1903.11314, 2019. [Online]. Available: http://arxiv.org/abs/1903.11314

[51] S. Merity, C. Xiong, J. Bradbury, and R. Socher, "Pointer sentinel mixture models," in *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*, 2017. [Online]. Available: https://openreview.net/forum?id=Byj72udxe

[52] P. Micikevicius, S. Narang, J. Alben, G. F. Diamos, E. Elsen, D. García, B. Ginsburg, M. Houston, O. Kuchaiev, G. Venkatesh, and H. Wu, "Mixed precision training," in *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*, 2018. [Online]. Available: https://openreview.net/forum?id=r1gs9JgRZ

[53] I. Micron Technology, "DDR4 power calculator 4.0," https://www.micron.com/~/media/documents/products/power-calculator/ddr4_power_calc.xlsm.

[54] H. Mostafa and X. Wang, "Parameter efficient training of deep convolutional neural networks by dynamic sparse reparameterization," in *International Conference on Machine Learning*, 2019, pp. 4646–4655.

[55] H. Mostafa and X. Wang, "Parameter efficient training of deep convolutional neural networks by dynamic sparse reparameterization," in *International Conference on Machine Learning*, 2019, pp. 4646–4655.

[56] NVIDIA, "Training with mixed precision," https://docs.nvidia.com/deeplearning/sdk/mixed-precision-training/index.html.

[57] S. Pal, J. Beaumont, D. . H. Park, A. Amarnath, S. Feng, C. Chakrabarti, H. . S. Kim, D. T. Blaauw, T. N. Mudge, and R. G. Dreslinski, "Outerspace: An outer product based sparse matrix multiplication accelerator," in *IEEE International Symposium on High Performance Computer Architecture, HPCA 2018, Vienna, Austria, February 24-28, 2018*. IEEE Computer Society, 2018, pp. 724–736. [Online]. Available: https://doi.org/10.1109/HPCA.2018.00067

[58] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S. W. Keckler, and W. J. Dally, "SCNN: an accelerator for compressed-sparse convolutional neural networks," in *Intl' Symp. on Computer Architecture*, ser. ISCA '17, 2017.

[59] R. Prabhakar, Y. Zhang, D. Koeplinger, M. Feldman, T. Zhao, S. Hadjis, A. Pedram, C. Kozyrakis, and K. Olukotun, "Plasticine: A reconfigurable architecture for parallel patterns," in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, June 2017, pp. 389–402.

[60] E. Qin, A. Samajdar, H. Kwon, V. Nadella, S. Srinivasan, D. The, B. Kaul, and T. Krishna, "SIGMA: A sparse and irregular GEMM accelerator with flexible interconnects for DNN training," in *IEEE International Symposium on High Performance Computer Architecture, HPCA 2020, San Diego, CA, USA, February 22-26, 2020*. IEEE, 2020, pp. 58–70. [Online]. Available: https://doi.org/10.1109/HPCA47549.2020.00015

[61] R. Rahman, *Intel Xeon Phi Coprocessor Architecture and Tools: The Guide for Application Developers*, 1st ed. Berkely, CA, USA: Apress, 2013.

[62] 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 *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2018, pp. 78–91.

[63] 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," *CoRR*, vol. abs/1409.0575, Sep. 2014.

[64] F. Schuiki, M. Schaffner, F. K. Gürkaynak, and L. Benini, "A scalable near-memory architecture for training deep neural networks on large in-memory datasets," *IEEE Transactions on Computers*, vol. 68, no. 4, pp. 484–497, April 2019.

[65] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[66] J. Snoek, H. Larochelle, and R. P. Adams, "Practical bayesian optimization of machine learning algorithms," in *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 2*, ser. NIPS'12, 2012, p. 2951–2959.

[67] E. Strubell, A. Ganesh, and A. McCallum, "Energy and policy considerations for deep learning in NLP," *CoRR*, vol. abs/1906.02243, 2019. [Online]. Available: http://arxiv.org/abs/1906.02243

[68] X. Sun, X. Ren, S. Ma, and H. Wang, "meProp: Sparsified back propagation for accelerated deep learning with reduced overfitting," in *Proceedings of the 34th International Conference on Machine Learning - Volume 70*, ser. ICML'17. JMLR.org, 2017, pp. 3299–3308. [Online]. Available: http://dl.acm.org/citation.cfm?id=3305890.3306022

[69] S. Venkataramani, A. Ranjan, S. Banerjee, D. Das, S. Avancha, A. Jagannathan, A. Durg, D. Nagaraj, B. Kaul, P. Dubey, and A. Raghunathan, "ScaleDeep: A scalable compute architecture for learning and evaluating deep networks," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ser. ISCA '17. New York, NY, USA: ACM, 2017, pp. 13–26. [Online]. Available: http://doi.acm.org/10.1145/3079856.3080244

[70] O. Vinyals, A. Toshev, S. Bengio, and D. Erhan, "Show and Tell: Lessons learned from the 2015 MSCOCO image captioning challenge," *CoRR*, vol. abs/1609.06647, 2016. [Online]. Available: http://arxiv.org/abs/1609.06647

[71] A. Wang, A. Singh, J. Michael, F. Hill, O. Levy, and S. R. Bowman, "GLUE: A multi-task benchmark and analysis platform for natural language understanding," 2019, in the Proceedings of ICLR.

[72] N. Wang, J. Choi, D. Brand, C.-Y. Chen, and K. Gopalakrishnan, "Training deep neural networks with 8-bit floating point numbers," in *Proceedings of the 32Nd International Conference on Neural Information Processing Systems*, ser. NIPS'18. USA: Curran Associates Inc., 2018, pp. 7686–7695. [Online]. Available: http://dl.acm.org/citation.cfm?id=3327757.3327866

[73] S. Wang and P. Kanwar, "BFloat16: The secret to high performance on cloud TPUs," 2019. [Online].

Available: https://cloud.google.com/blog/products/ai-machine-learning/bfloat16-the-secret-to-high-performance-on-cloud-tpus

[74] W. Wen, C. Xu, F. Yan, C. Wu, Y. Wang, Y. Chen, and H. Li, "TernGrad: Ternary gradients to reduce communication in distributed deep learning," in *Advances in Neural Information Processing Systems 30*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, Eds. Curran Associates, Inc., 2017, pp. 1509–1519. [Online]. Available: http://papers.nips.cc/paper/6749-terngrad-ternary-gradients-to-reduce-communication-in-distributed-deep-learning.pdf

[75] Y. Wu, A. Kirillov, F. Massa, W.-Y. Lo, and R. Girshick, "Detectron2," https://github.com/facebookresearch/detectron2, 2019.

[76] A. Yang, "Deep learning training at scale spring crest deep learning accelerator (intel® nervana™ NNP-T)," in *2019 IEEE Hot Chips 31 Symposium (HCS), Cupertino, CA, USA, August 18-20, 2019*, 2019, pp. 1–20. [Online]. Available: https://doi.org/10.1109/HOTCHIPS.2019.8875643

[77] D. Zhang, J. Yang, D. Ye, and G. Hua, "LQ-Nets: Learned quantization for highly accurate and compact deep neural networks," in *Proceedings of the European Conference on Computer Vision (ECCV)*, 2018, pp. 365–382.

[78] J. Zhang, X. Chen, M. Song, and T. Li, "Eager pruning: Algorithm and architecture support for fast training of deep neural networks," in *Proceedings of the 46th International Symposium on Computer Architecture*, ser. ISCA '19. New York, NY, USA: ACM, 2019, pp. 292–303. [Online]. Available: http://doi.acm.org/10.1145/3307650.3322263

[79] S. Zhang, Z. Du, L. Zhang, H. Lan, S. Liu, L. Li, Q. Guo, T. Chen, and Y. Chen, "Cambricon-X: An accelerator for sparse neural networks," in *Intl' Symp. on Microarchitecture*, 2016. [Online]. Available: https://doi.org/10.1109/MICRO.2016.7783723

[80] S. Zhang, Z. Du, L. Zhang, H. Lan, S. Liu, L. Li, Q. Guo, T. Chen, and Y. Chen, "Cambricon-X: An accelerator for sparse neural networks," in *Intl' Symp. on Microarchitecture*, 2016.

[81] B. Zheng, N. Vijaykumar, and G. Pekhimenko, "Echo: Compiler-based GPU memory footprint reduction for LSTM RNN training," in *Proceedings of the 47th Annual International Symposium on Computer Architecture*, ser. ISCA '20. ACM, 2020.

[82] 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 *Intl' Symp. on Microarchitecture*, 2018.