FPRAker: A Processing Element For Accelerating Neural Network Training

Omar Mohamed Awad*, Mostafa Mahmoud*, Isak Edo*, Ali Hadi Zadeh*, Ciaran Bannon*, Anand Jayarajan*
Gennady Pekhimenko*, Andreas Moshovos*
{omar.awad, mostafa.mahmoud, isak.edo, a.hadizadeh, ciaran.bannon}@mail.utoronto.ca,
{anandj, pekhimenko}@cs.toronto.edu, moshovos@ece.utoronto.ca
* University of Toronto, † Vector Institute

Abstract—We present FPRAker, a processing element for composing training accelerators. FPRAker processes several floating-point multiply-accumulate operations concurrently and accumulates their result into a higher precision accumulator. FPRAker boosts performance and energy efficiency during training by taking advantage of the values that naturally appear during training. It processes the significant of the operands of each multiply-accumulate as a series of signed powers of two. The conversion to this form is done on-the-fly. This exposes ineffectual work that can be skipped: values when encoded have few terms and some of them can be discarded as they would fall outside the range of the accumulator given the limited precision of floating-point. FPRAker also takes advantage of spatial correlation in values across channels and uses delta-encoding off-chip to reduce memory footprint and bandwidth. We demonstrate that FPRAker can be used to compose an accelerator for training and that it can improve performance and energy efficiency compared to using optimized bit-parallel floating-point units under iso-compute area constraints. We also demonstrate that FPRAker delivers additional benefits when training incorporates pruning.

I. MOTIVATION

The pervasive applications of deep learning and the end of Dennard scaling have been driving efforts for accelerating deep learning inference and training. These efforts span the full system stack, from algorithms, to middleware and hardware architectures. Here we target training, a task that includes inference as a subtask. Training is a compute- and memory-intensive task often requiring weeks of compute time even with state-of-the-art training methods and hardware [1]. The cost of training is staggering. For example, training XNets requires 512 TPU v3 chips for 2.5 days leading to a total training cost above $61K [2]. Further, training Grover-Mega requires 128 TPU v3 chips for 2 weeks leading to a total training cost of $25K [3]. Often multiple trials are required for hyperparameter tuning which further amplifies the cost.

During training a set of annotated inputs, that is inputs for which the desired output is known is processed by repeatedly performing a forward and backward pass. The forward pass performs inference whose output is initially inaccurate. However, given that the desired outputs are known, the training algorithm can calculate a loss, a metric of how far the outputs are from the desired ones. During the backward pass, this loss is used to adjust the network’s parameters and to have it slowly converge to its best possible accuracy.

Numerous methods have been developed to accelerate training, and fortunately often they can be used in combination. Distributed training partitions the training workload across several computing nodes taking advantage of data, model, or pipeline parallelism [4], [5], [6], [7], [8]. Timing communication and computation can further reduce training time [9], [10], [11], [12], [13]. Dataflow optimizations to facilitate data blocking and to maximize data reuse reduces the cost of on- and off-chip accesses within the node maximizing reuse from lower cost components of the memory hierarchy [14], [15]. Another family of methods reduces the footprint of the intermediate data needed during training. For example, in the simplest form of training, all neuron values produced during the forward pass are kept to be used during backpropagation. Batching and keeping only one or a few samples instead reduces this cost. Lossless and lossy compression methods further reduce the footprint of such data [9], [16], [17], [18], [19], [20]. Finally, selective backpropagation methods alter the backward pass by propagating loss only for some of the neurons [21] thus reducing work.

On the other hand, the need to boost energy efficiency during inference has led to techniques that increase computation and memory needs during training. This includes works that perform network pruning and quantization during training. Pruning zeroes out weights and thus creates an opportunity for reducing work and model size during inference. Quantization produces models that use shorter and more energy efficient to compute with datatypes such as 16b, 8b or 4b fixed-point values. Parameter Efficient Training [22], Memorized Sparse Back-propagation [23] are examples of recent pruning methods. PACT [24] and outlier-aware quantization [25] are training time quantization methods. Network architecture search techniques also increase training time as they adjust the model’s architecture [26].

Despite the successes already reported via the aforementioned methods, and while we expect that these methods will be refined further, the need to further accelerate training both at the data center and at the edge remains unabated. Operating and maintenance costs, latency, throughput, and node count are major considerations for data centers. At the edge energy and latency are major considerations where training may be primarily used to refine or augment already trained models [27].

*This version of the paper was submitted to MICRO2020. A previous version was submitted to ISCA2020.
Regardless of the target application, improving node performance would be of value. Accordingly, in this work we target methods that could complement existing training acceleration methods. In general, the bulk of the computations and data transfers during training is for performing multiply-accumulate operations (MAC) during the forward and backward passes. As mentioned above, compression methods can greatly reduce the cost of data transfers. Here we target the processing elements for these operations and propose designs that exploit ineffectual work that occurs naturally during training and whose frequency is amplified by quantization, pruning, and selective backpropagation.

To devise a processing element that can eliminate ineffectual work during training we may attempt to build upon the numerous proposals that exploit ineffectual work during inference. We highlight some those approaches that are most relevant to this work. Any MAC operation where any of the two input operands, be it the activation or the weight is ineffectual. The first class of accelerators rely on that zeros occur naturally in the activations of many models especially when they use ReLU [30], [31], [32], [33]. There are several accelerators that target pruned models e.g., [31], [34], [35], [36], [37], [38], [39], [40], [41], [42]. Another class of designs benefit from reduced value ranges whether these occur naturally or result from quantization. This includes bit-serial designs [43], [44], [45], [46], [47], and designs that support many different datatypes such as BitFusion [48]. Finally, another class of designs targets bit-sparsity where by decomposing multiplication into a series of shift-and-add operations they expose ineffectual work at the bit-level [38], [49], [50].

While we can draw from the experience gained from the aforementioned design for inference, training presents us with different challenges. First, is the datatype. While models during inference work with fixed-point values of relatively limited range, the values training operates upon tend to be spread over a large range. Accordingly, training implementations use floating-point arithmetic with single-precision IEEE floating point arithmetic (FP32) being sufficient for virtually all models. Other datatypes that facilitate the use of more energy- and area-efficient multiply-accumulate units compared to FP32 have been successfully used in training many models. These include bfloat16, and 8b or smaller floating-point formats [51], [52], [53], [54], [55], [56], [57]. Moreover, since floating-point arithmetic is a lot more expensive than integer arithmetic, mixed datatype training methods use floating-point arithmetic only sparingly [55], [58], [59], [60]. Despite these proposals, FP32 remains today the standard fall-back format, especially for training on large and challenging datasets. As a result of its limited range and the lack of an exponent, the fixed-point representation used during inference gives rise to zero values (too small a value to be represented), zero bit prefixes (small value that can be represented), and bit sparsity (most values tend to be small and few are large) that the aforementioned inference accelerators rely upon. FP32 can represent much smaller values, its mantissa is normalized, and whether bit sparsity exists has not been demonstrated.

Second, is the computation structure. Inference operates on two tensors, the weights and the activations, performing per layer a matrix/matrix or matrix/vector multiplication or pairwise vector operations to produce the activations for the next layer in a feed-forward fashion. Training includes this computation as its forward pass which is followed by the backward pass that involves a third tensor, the gradients. Most importantly, the backward pass uses the activation and weight tensors in a different way than the forward pass, making it difficult to pack them efficiently in memory, more so to remove zeros as done by inference accelerators that target sparsity. Related to computation structure, third, is value mutability and value content. Whereas in inference the weights are static, they are not so during training. Furthermore, training initializes the network with random values which it then slowly adjusts. Accordingly, one cannot necessarily expect the values processed during training to exhibit similar behavior such as sparsity or bit-sparsity. More so, for the gradients which are values that do not appear at all during inference.

Our first contribution is that we demonstrate that a large fraction of the work performed during training is ineffectual. To expose this ineffectual work we decompose each multiplication into a series of single bit multiply-accumulate operations. This reveals two sources of ineffectual work: First, more than 85% of the computations are ineffectual since one of the inputs is zero. Second, the combination of the high dynamic range (exponent) and the limited precision (mantissa) often yields values which are non-zero, yet too small to affect the accumulated result, even when using extended precision (e.g., trying to accumulate $2^{-64}$ into $2^{64}$).

This observation led us to consider whether it is possible to use bit-skipping (bit-serial where we skip over zero bits) processing to exploit these two behaviors. Bit-skipping processing of multiply-accumulate operations has been proposed before for inference. Bit-Pragmatic is a data-parallel processing element that performs such bit-skipping of one operand side [38] whereas Laconic does so for both sides [50]. Since these methods target inference only they work with fixed-point values. Since we found that there is little bit-sparsity in the weights during training, we focused on Bit-Pragmatic’s approach. Converting a fixed-point design to floating-point is a non-trivial task to start with. Regardless, converting Bit-Pragmatic into floating-point resulted in an area-expensive unit which performs poorly under iso-compute area constraints. Specifically, compared to an optimized Bfloat16 processing element (see Section V-A) that performs 8 MAC operations, under iso-compute constraints, an optimized accelerator configuration using the Bfloat16 Bit-Pragmatic PEs is on average $1.72 \times$ slower and $1.96 \times$ less energy efficient. In the worst case, the Bfloat16 bit-pragmatic PE was $2.86 \times$ slower and $3.2 \times$ less energy efficient. The Bfloat16 Bit-Pragmatic PE is $2.5 \times$ smaller than the bit-parallel PE, and while we can use more such PEs for the same area, we cannot fit enough of them to boost performance via parallelism as required by all bit-serial and bit-skipping designs.

Accordingly, our second contribution is FPRaker, a processing tile for training accelerators which exploits both bit-sparsity and out-of-bounds computations. FPRaker comprises several adder-tree based processing elements organized in
a grid so that it can exploit data reuse both spatially and temporally. The processing elements multiply multiple value pairs concurrently and accumulate their products into an output accumulator. They process one of the input operands per multiplication as a series of signed powers of two hitherto referred to as *terms*. The conversion of that operand into powers of two is performed on the fly; all operands are stored in floating point form in memory. The processing elements take advantage of ineffectual work that stems either from mantissa bits that were zero or from out-of-bounds multiplications given the current accumulator value. The tile is designed for area efficiency and it incorporates the following designs choices for this purpose: a) The processing element limits the range of powers-of-two that they can be processed simultaneously greatly reducing the cost of its shift-and-add components. b) A common exponent processing unit that is time-multiplexed among multiple processing elements. c) The power-of-two encoders are shared along the rows. d) Per processing element element buffers reduce the effects of work imbalance across the processing elements. e) The PE implements a low cost mechanism for eliminating out-of-range intermediate values. Skipping out-of-bound intermediate values not only reduces the amount of work, but more importantly proves very effective in reducing the effect of cross-lane synchronization. Section III-A explains that a bit-parallel unit could only take advantage of these values to power-gate some of its component but not to also improve performance.

In more detail, *FPRaker* has the following characteristics: a) It does not affect numerical accuracy. The results it produces adhere to the floating-point arithmetic used during training. b) It skips ineffectual operations that would result from zero mantissa bits and from out-of-range intermediate values. c) Despite individual MAC operations taking more than one cycle, *FPRaker*’s computational throughput is higher compared to conventional floating-point units; given that *FPRaker* processing elements are much smaller we can fit more of them in the same area. c) It naturally supports shorter mantissa lengths thus rewarding innovation in training methods with mixed or shorter datatypes [54], [61]. It does so while not requiring that the methods be universally applicable to all models. d) *FPRaker* allows us to choose which tensor input we wish to process serially per layer. This allows us to target those tensors that have more sparsity depending on the layer and the pass (forward or backward).

Our third contribution is a simple, low-overhead memory encoding for floating-point values that relies on the value distribution that is typical in deep learning training. We observed that consecutive values across channels have similar values and thus exponents. Accordingly, we encode the exponents as deltas for groups of such values. We use this encoding when storing and reading values off chip, thus further reducing the cost of memory transfers.

We highlight the following experimental observations: a) While some neural networks naturally exhibit zero values (sparsity) during training, unless pruning is used, this is limited to the activations and the gradients. b) term-sparsity exists in all tensors including the weights and is much higher than sparsity.

c) Compared to an accelerator using optimized bit-parallel FP32 processing elements and that can perform 4K bfloat16 [52], [53] MACs per cycle, a configuration that uses the same compute area to deploy *FPRaker* PEs is 1.5× faster and 1.4× more energy efficient. d) Performance benefits with *FPRaker remain fairly stable throughout the training process for all three major operations. e) *FPRaker* can be used in conjunction with training methods that specify a different accumulator precision to be used per layer. There it can improve performance vs using an accumulator with a fixed width significant by 38% for ResNet18.

### II. INEFFECTUAL WORK IN TRAINING

This section motivates *FPRaker* by measuring the work reduction that is theoretically possible with two related approaches: 1) by removing all MACs where at least one of the operands are zero (value sparsity, or simply sparsity), and 2) by processing only the non-zero bits of the mantissa of one of the operands (bit sparsity). We study the performance of *FPRaker* on a wide variety of applications. Table I lists the models studied. ResNet18-Q is a variant of ResNet18 trained using PACT [24] which quantizes both activations and weights down to 4b during training. ResNet50-S2 is a variant of ResNet50 trained using dynamic sparse reparameterization [62] which targets sparse learning that maintain high weight sparsity throughout the training process while achieving accuracy levels comparable to baseline training. SNLI performs natural language inference and comprises of fully-connected, LSTM-encoder, ReLU, and dropout layers. Image2Text is an encoder-decoder model for image-to-markup generation. We study three models of different tasks from MLPerf training benchmark: 1) Detectron2: an object detection model based on Mask R-CNN, 2) NCF: a model for collaborative filtering, and 3) Bert: a transformer-based model using attention. For our measurements we sample one randomly selected batch per epoch over as many epochs as necessary to train the network to its originally reported accuracy (up to 90 epochs were enough for all).

The bulk of work during training is due to three major operations per layer:

\[ Z = I \cdot W \]  
\[ E = W^T \frac{\partial E}{\partial Z} \]  
\[ I = \frac{\partial E}{\partial W} = I \cdot \frac{\partial E}{\partial Z} \]  

For convolutional layers Eq. 1 describes the convolution of activations (I) and weights (W) that produces the output activations (Z) during forward propagation. There the output Z passes through an activation function before used as input for the next layer. Eq. 2 and 3 describe the calculation of the activation \( \frac{\partial E}{\partial I} \) and weight \( \frac{\partial E}{\partial W} \) gradients respectively in
the backward propagation. Only the activation gradients are back-propagated across layers. The weight gradients update the layer’s weights once per batch. For fully-connected layers the equations describe several matrix-vector operations. For other operations they describe vector operations or matrix-vector operations. For clarity, in the rest of this work we will refer to the gradients as $G$.

Fig. 1a, and 1b show the value, and term-sparsity respectively and for each of the three tensors ($W$, $I$, and $G$). Each value is weighted according to frequency of use. We use the term term-sparsity to signify that for these measurements the mantissa is first encoded into signed powers of two using Canonical-encoding which is a variation of Booth-encoding. This is because of the bit-serial processing of the mantissa.

The activations in the image classification networks exhibit sparsity exceeding 35% in all cases. This is expected since these networks use the ReLU activation function which clips negative values to zero. However, Weight sparsity is typically low and only some of the classification models exhibit sparsity in their gradients. For the remaining models, however, such as those for natural language processing, value sparsity is very low for all three tensors. Regardless, since some models do exhibit sparsity it may be worthwhile to investigate whether it possible to exploit it during training. As explained in the introduction, this is a non-trivial task, as training is different for all models regardless of the target application. Given that term-sparsity is more prevalent than value sparsity, and exists in all models in the rest of this work we investigate whether it is practically possible to exploit it during training. One such option is the FPRaker processing element which we present here.

Fig. 1b shows that all three tensors exhibit high term-sparsity for all models regardless of the target application. Given that term-sparsity is more prevalent than value sparsity, and exists in all models in the rest of this work we investigate whether it is practically possible to exploit it during training. One such option is the FPRaker processing element which we present next.

Fig. 2 reports the ideal potential speedup due to reduction in the multiplication work through skipping the zero terms in the serial input. We calculate the potential speedup over the baseline as:

$$\text{Potential speedup} = \frac{\#\text{MAC operations}}{\text{term sparsity} \times \#\text{MAC operations}} \quad (4)$$

III. FPRaker Approach

FPRaker’s goal is to take advantage of bit sparsity in one of the operands used in the three operations performed during training (equations 1 through 3) all of which are composed of many MAC operations. We first explain how decomposing MAC operations into a series of shift-and-add operations can expose ineffectual work, providing us with the opportunity to save energy and time. Section IV-A then describes the FPRaker processing element and Section IV-C details the FPRaker tile.

A. Exposing Ineffectual Work

To expose ineffectual work during MAC operations we can decompose the operation into a series of “shift and add” operations, Let us first look at the multiplication. Let $A = 2^{A_e} \times A_m$ and $B = 2^{B_e} \times B_m$ be two values in floating point, both represented as an exponent ($A_e$ and $B_e$) and a significant ($A_m$ and $B_m$) which is normalized and includes the implied “1.”. Conventional floating point units perform this multiplication in a single step (sign bits are XORed):

$$A \times B = 2^{A_e+B_e} \times (A_m \times B_m) = (A_m \times B_m) \ll (A_e + B_e) \quad (5)$$

By decomposing $A_m$ into a series of signed powers of two $A_m^p$ where $A = \sum_{i=0}^{p} A_m^p$ and $A_m^p = \pm 2^i$, we can instead perform the multiplication as follows:

$$A \times B = (\sum_{i=0}^{p} A_m^p \times B_m) \ll (A_e + B_e) = \sum_{i=0}^{p} B_m \ll (i + A_e + B_e) \quad (6)$$

For example, if $A_m = 1.0000001b$, $A_e = 10b$, $B_m = 1.1011011b$, $B_e = 11b$ then we can perform $A \times B$ as two shift-and-add operations of $B_m \ll (10b+11b-10b)$ and $B_m \ll (10b+00bin 11b-11b)$. A conventional multiplier would process all bits of $A_m$ despite performing ineffectual work for the six bits that are zero.

However, this decomposition exposes further ineffectual work that conventional units perform as a result of the high dynamic range of values that floating point seeks to represent. Informally, some of the work done during the multiplication will result in values that will be out-of-bounds given the accumulator value. To understand why this is the case let us now consider not only the multiplication but also the accumulation. Let’s assume that the product $A \times B$ will be accumulated into a running sum $S$ and $S_e$ is much larger than $A_e + B_e$. It will not possible to represent the sum $S + A \times B$ given the limited precision of the mantissa. In other cases, some of the “shift-and-add” operations would be guaranteed to fall outside the mantissa even when we consider the increased mantissa length used.
to perform rounding, i.e., partial swamping. A conventional pipelined MAC unit can at best power-gate the multiplier and accumulator after comparing the exponents and only when the whole multiplication result falls out of range. However, it cannot use this opportunity to reduce cycle count. By decomposing the multiplication into several simpler operations, we can terminate the operation in a single cycle given that we process the bits from the most to the least significant, and thus boost performance by initiating another MAC earlier. The same is true when processing multiple $A \times B$ products in parallel in an adder-tree processing element. A conventional adder-tree based MAC unit can potentially power-gate the multiplier and the adder tree branches corresponding to products that will be out-of-bounds. The cycle will still be consumed. As we explain in the next section, a shift-and-add based unit will be able to terminate such products in a single cycle and advance others in their place.

IV. FPRaker Implementation

Section IV-A describes the design of an FPRaker processing element. Section IV-B explains how FPRaker time-multiplexes a single exponent block among multiple PEs, a key optimization for area- and energy-efficiency. Section IV-C and Section IV-E explain respectively how multiple processing elements can be organized into a larger tile, and how data is organized in memory so that it can be supplied to the processing elements to keep them busy.

A. FPRaker Processing Element

The FPRaker PE performs the multiplication of 8 Bfloat16 ($A, B$) value pairs, concurrently accumulating the result into an output accumulator. The Bfloat16 format consists of a sign bit, followed by a biased 8b exponent, and a normalized 7b significand (mantissa). Fig. 3 shows a baseline of the FPRaker PE design which performs the computation in 3 blocks: exponent, reduction, and accumulation. We describe an implementation where the 3 blocks are performed in a single cycle. We will build upon this design and modify it to construct a more area efficient tile comprising several of these PEs. Recall that the significands of each of the $A$ operands are converted on-the-fly into a series of terms (signed powers of two) using canonical encoding, e.g. $A=(1.1110000)$ is encoded as $(+2^{1}, -2^{-4})$. This encoding occurs just before the input to the PE. All values stay in bfloat16 while in memory. The PE will process the $A$ values term-serially. The accumulator has an extended 13b significand; 1b for the leading 1 (hidden), 9b for extended precision following the chunk-based accumulation scheme as suggested by Sakr et al., [69] with a chunk-size of 64, plus 3b for rounding to nearest even. It has 3 additional integer bits following the hidden bit so that it can fit the worst case carry out from accumulating 8 products. In total the accumulator has 16b, 4 integer, and 12 fractional.

The PE accepts 8 8-bit $A$ exponents $A_{0}, ..., A_{7}$, their corresponding 8 3-bit significand terms $t_{0}, ..., t_{7}$ (after canonical encoding) and signs bits $A_{0}, ..., A_{7}$, along with 8 8-bit $B$ exponents $B_{e0}, ..., B_{e7}$, their significands $B_{m0}, ..., B_{m7}$ (as-is) and their sign bits $B_{e0}, ..., B_{e7}$ as shown in Fig. 3.

**Block 1 — Exponent:** Processing a new set of 8 value pairs starts first at the exponent block. This block adds the $A$ and $B$ exponents in pairs to produce the exponents $A B e_{i}$ for the corresponding products. A comparator tree (MAX) takes these product exponents and the exponent of the accumulator and calculates the maximum $e_{\text{max}}$. The maximum exponent is used to align all products so that they can be summed correctly. To determine the proper alignment per product the block subtracts all product exponents from $e_{\text{max}}$ calculating the alignment offsets $\Delta e_{i}$. The maximum exponent is used to also discard terms that will fall out of-bounds when accumulated. The PE will skip any terms who fall outside the $e_{\text{max}}-12$ range. The block is invoked only once per new set of value pairs, before any terms are generated, and regardless of how many terms end up being generated. Accordingly, the minimum effective number of cycles for processing the 8 MACs will be 1 cycle regardless of value (the blocks can be pipelined, and since there are no data dependencies, the pipeline can be kept full).

In case one of the resulting products has an exponent larger than the current accumulator exponent, the accumulator will be shifted accordingly prior to accumulation (acc_shift signal).

**Block 2 — Shift&Reduce:** Since multiplication with a term amounts to shifting, this block calculates the number of bits by which each $B$ significand will have to be shifted by prior to accumulation. These are the 4-bit terms $K_{0}, ..., K_{7}$. To calculate $K_{i}$ we add the product exponent deltas ($\Delta e_{i}$) to the corresponding $A$ term $t_{i}$. The $A$ sign bits are XORed with their corresponding $B$ sign bits to determine the signs of the products $P_{0}, ..., P_{7}$. The $B$ significands are complemented according to their corresponding product signs, and then shifted using the offsets $K_{0}, ..., K_{7}$. The PE uses a shifter per $B$ significand to implement the multiplication. In contrast, a conventional floating point unit would require shifters at the output of the
multiplier. Thus FPRAker PE effectively completely eliminates the cost of the multipliers.

Bits that are shifted out of the accumulator range from each \( B_m \) operand are rounded using round-to-nearest-even (RNE) approach. An adder tree reduces the 8 \( B_m \) operands into a single partial sum as shown in Fig. 3(2).

Figure 6: Exponent histogram of layer Conv2d_8 in epochs 0 and 89 of training ResNet34 on ImageNet. The figure shows only the utilized part of the full range [-127:128] of an 8b exponent.

**Block 3 — Accumulation:** The resulting partial sum from step 2 is added to the correctly aligned value of the accumulator register. In each accumulation step, the accumulator register is normalized and rounded using the rounding-to-nearest-even (RNE) scheme. The normalization block updates the accumulator exponent as shown in Fig. 3(3). When the accumulator value is read out, it is converted to bfloat16 by extracting only 7b for the significand.

In the worst case two \( K_i \) offsets may differ by up to 12 since our accumulator has 12 fractional bits. This means that the baseline PE requires relatively large shifters and an accumulator tree that accepts wide inputs. Specifically, the PE requires shifters that can shift up to 12 positions a value that is 8b (7b significant + hidden bit). Had this been integer arithmetic we would need to accumulate 12+8=20b wide. However, since this is a floating point unit we will be accumulating only the 14 most significant bits (1b hidden, 12b fractional and the sign). Any bits falling below this range will be included in the sticky bit which is the least significant bit of each input operand.

It is possible to further reduce this cost by taking advantage of the expected distribution of the exponents. Fig. 6 shows, for example, the distribution of exponents for a layer of ResNet34. The vast majority of the exponents of the inputs, the weights and the output gradients lie within a narrow range. This suggests that in the common case the exponent deltas will be relatively small. In addition, the MSBs of the activations are guaranteed to be one (given denormals are not supported [53]). This indicates that very often the \( K_0, \ldots, K_7 \) offsets would lie within a narrow range. We take advantage of this behavior to reduce the PE area. In our preferred configuration we limit the maximum difference in among the \( K_i \) offsets that can be handled in a single cycle to be up to \( 3 \). As a result, the shifters need to support shifting by up to \( 3b \) and the adder tree now needs to process \( 12b \) inputs (1b hidden, 7b+3b significant, and the sign bit). A shared single shifter (base_shift) after the adder tree serves a dual purpose: First, it aligns the adder tree’s output and the accumulator properly. Second, it allows the PE to skip over longer than \( 3b \) distances in the input term stream. This is useful, when the next set of terms are at a distance longer than \( 3b \) vs. the current ones. Fig. 4 shows the modified PE. Each PE has a control unit to generate the modified terms \( \Delta_i \) and a valid signal indicating whether the lane can process its term at the current cycle.

 Skipping out-of-bounds terms turns out to be surprisingly inexpensive. The control unit uses a comparator per lane to check if its current \( K \) term lies within a threshold with the value of the accumulator precision (comparators are optimized by the synthesis tool for comparing with a constant) and feeds back an \( OB \) signal to its corresponding term encoder indicating that any subsequent term coming from the same input pair is guaranteed to be ineffectual (out-of-bound) term. Hence, FPRAker can boost its performance and energy-efficiency by skipping the
processing of the subsequent out-of-bound terms. The $OB_i$ signals of a certain lane across the PEs of the same tile column are synchronized together. The threshold is currently set according to [69] which ensures models converge within 0.5% of the FP32 training accuracy on ImageNet dataset. However, the threshold can be controlled effectively implementing a dynamic bit-width accumulator which can boost performance by increasing the number of skipped “out-of-bounds” bits, an option we do not investigate further. Fig. 5 shows an example of a simplified PE processing 2 activation-weight pairs each with 4b mantissa: $A_0 = 2^5 \times 1.1101, B_0 = 2^3 \times 1.0011$ and $A_1 = 2^1 \times 1.1011, B_1 = 2^1 \times 1.1010$.

**B. Sharing the Exponent Block**

In the common case, processing a group of $A$ values will require multiple cycles since some of them will be converted into multiple terms. During that time the inputs to the exponent block will not change. To further reduce area we can take advantage of this expected behavior and share the exponent block across multiple PEs. The decision of how many PEs to share an exponent block over can be based on the expected bit-sparsity. The lower the bit-sparsity then higher the processing time per PE and the less often it will need a new set of exponents. Hence, the more the PEs that can share the exponent block. Since the studied models are highly sparse, sharing one exponent block per two PEs proved best. Figure 7 shows the modified design. The unit as a whole accepts as input one exponent block per two PEs proved best. Figure 7 shows the modified design. The unit as a whole accepts as input one set of 8 $A$ inputs and two sets of $B$ inputs, $B$ and $B'$. The exponent block can process one of $(A, B)$ or $(A, B')$ at a time. During the cycle when it processes $(A, B)$, the multiplexer for $PE\#1$ passes on the $e_{max}$ and exponent deltas directly to the PE. Simultaneously, these values will be latched into the registers in front of the PE so that they remain constant while the PE processes all terms of input $A$. When the exponent block processes $(A, B')$ the aforementioned process proceeds with $PE\#2$. With this arrangement both PEs must finish processing all $A$ terms before they can proceed to process another set of $A$ values. Since the exponent block is shared, each set of 8 $A$ values will take at least 2 cycles to be processed (even if it contains zero terms).

**C. FPRaker Tile**

By utilizing per PE buffers it is possible to exploit data reuse temporally. To exploit data reuse spatially we can arrange several PEs into a tile. Fig. 8 shows an example of a 2 x 2 tile of PEs and each PE performs 8 MAC operations in parallel. Each pair of PEs per column shares an exponent block as previously described. The $B$ and $B'$ inputs are shared across PEs in the same row. For example, during the forward pass we can have different filters being processed by each row and different windows processed across the columns. Since the $B$ and $B'$ inputs are shared all columns would have to wait for the column with the most $A$, terms to finish before advancing to the next set of $B$ and $B'$ inputs. To reduce these stalls the tile introduces per $B$ and $B'$ buffers. By having $N$ such buffers per PE allows the columns be at most $N$ sets of values ahead.

**D. Exponent Base-Delta Compression**

Motivated by the narrow value distribution shown in Fig. 6, we studied the spatial correlation of values during training. We found that consecutive values across all dimensions (channel, H, or W) have similar values. This is true for the activations, the weights and the output gradients. Similar values in floating-point have similar exponents, a property which we can exploit through a base-delta compression scheme [70]. In our experiments, we block values channel-wise (we present evidence, however, that the value correlation persists along the H dimension as well) into groups of 32 values each, where the exponent of the first value in the group is the base and we compute the delta exponent for the rest of the values in the group relative to it as shown in Fig. 9. The bit-width ($\delta$) of the delta exponents is dynamically determined per group and is set to the maximum precision ($P$) of the resulting delta exponents per group similar to the approach of Delmas et al. [47]. The delta exponent bit-width (3b) is attached to the header of each group as metadata.

![Figure 7: Reducing area by sharing the exponent block between two PEs.](image)

![Figure 8: A 2 x 2 PE FPRaker Tile.](image)

![Figure 9: Group of 32 values with exponent base-delta compression](image)
Fig. 10 shows the total, normalized exponent footprint after the base-delta compression. Our technique is effective for both channel-wise and spatial (H dimension shown) dataflows. We use this compression scheme to reduce the off-chip memory bandwidth. Values are compressed at the output of each layer and before writing them off-chip, and they are decompressed when they are read back on-chip.

![Normalized Exponent Footprint](square of 32) 20% 40% 60% 80% 100%

Figure 10: Memory savings due to exponent base-delta compression. Bars and markers represent compression channel-wise and spatial-wise, respectively.

### E. Data Supply

Focusing solely on computation is insufficient. Data transfers account for a significant portion and often dominate energy consumption in deep learning. Accordingly, it is essential to consider what the memory hierarchy needs to do to keep the execution units busy. A challenge with training is that while it processes three arrays \( I \), \( W \) and \( G \) the order in which their elements are grouped differs across the three major computations (Eq. 1 through 3). However it is possible to rearrange the arrays as they are read from off-chip. For this purpose we store the arrays in memory using a container of “square” of 32×32 bfloat16 values. This a size that matches well the typical row sizes of DDR4 memories and allows us to achieve high bandwidth when reading values from off-chip. A container includes values from coordinates \((c, r, k)\) (channel, row, column) to \((c+31, r, k+31)\) where \(c\) and \(k\) are divisible by 32 (padding is used as necessary). Containers are stored in channel, column, row order. When read from off-chip memory the container values are stored either in the exact same order on the multibanked on-chip buffers. The tiles can then access data directly reading 8 bfloat16 values per access. The weights and the activation gradients however need to be processed in different order depending on the operation performed. Effectively, the respective arrays must be accessed in the transpose order during one of the operations. For this purpose we incorporate transposer units on-chip. A transposer reads in 8 blocks of 8 bfloat16 values from the on-chip memories. Each of these 8 reads uses 8-value wide reads as mentioned above and the blocks are written as rows in an internal to the transposer buffer. Collectively these blocks form an 8×8 block of values. The transposer can read out 8 blocks of 8 values each and send those to the processing units. Each of these blocks however is read out as a column from its internal buffer. This effectively transposes the 8×8 value group.

### V. Evaluation

This section evaluates the performance of FPRaker comparing against an equivalent baseline architecture that uses conventional floating-point units. The rest of this section is organized as follows: Section V-A presents the experimental methodology. Section V-C evaluates the performance of FPRaker over the baseline architecture. Section V-D reports the energy efficiency.

#### A. Methodology

A custom cycle-accurate simulator was developed to model the execution time of FPRaker and of the baseline architecture. Besides modeling timing behavior the simulator also models value transfers and computation in time faithfully and checks the produced values for correctness against the golden values. The simulator was validated with microbenchmarking. For area and power analysis, both FPRaker and the baseline designs were implemented in Verilog and synthesized using Synopsys’ Design Compiler [71] for 600 MHz clock frequency with a 65nm TSMC technology (due to licensing restrictions we cannot get access to a better technology) and with a commercial library for the given technology. We use Cadence Innovus [72] for layout generation. We use Intel’s PSG ModelSim to generate data-driven activity factors which we feed to Innovus to estimate the power. The baseline MAC unit was optimized for area, energy and latency. Generally, it is not possible to optimize for all three, however, in the case of MAC units, it is possible [73]. We use an efficient bit-parallel fused MAC unit as the baseline PE. The constituent multipliers are both area and latency efficient, and are taken from the DesignWare IP library developed by Synopsys. Further, we optimize the baseline units for deep learning training by reducing the precision of its I/O operands to bfloat16 and accumulating in reduced precision with chunk-based accumulation similar to FPRaker units. The area and energy consumption of the on-chip SRAM Global Buffer (GB) is divided into activation, weight, and gradient memories which were modeled using CACTI [74]. The Global Buffer has an odd number of banks to reduce bank conflicts for layers with a stride greater than one. To estimate the latency and energy consumption of the off-chip DRAM memory we use the model provided by Micron [75]. The configurations for both FPRaker and the baseline are shown in Table II.

| Tile Configuration | FPRaker | Baseline |
|--------------------|---------|----------|
| Tiles              | \(8 \times 8\) | \(8 \times 8\) |
| Total PEs          | 2304    | 512      |
| Multipliers/PE     | 8       | 8        |
| MACs/cycle         | -       | 4096     |
| Scratchpads        | 2KB     |          |
| Office DRAM Memory | 16GB 4-channel LPDDR4-3200 |          |
outputs for each layer using Pytorch Forward and Backward hooks. For BERT we traced BERT-base and the fine-tuning training for a GLUE task. The simulator uses the traces to model execution time and collects activity statistics so that energy can be modeled.

B. Area

Since FPRaker processes one of the inputs term-serially a single FPRaker processing engine can never outperform a conventional PE that processes the same number of inputs. FPRaker relies on parallelism to extract more performance. This is only possible if we can afford to use more FPRaker units than conventional units. One approach is to use an iso-compute area constraint. That is to determine how many FPRaker tiles we can fit in the same area for a baseline tile. For this we take into account only the compute cores as associated logic and not the scratchpads.

The conventional PE we compared against process concurrently 8 pairs of bfloat16 values and accumulates their sum. We can include buffers for the inputs (A and B) and the outputs so that we can exploit data reuse temporally. We can also organize multiple PEs in a grid sharing buffers and inputs across rows and columns to also exploit reuse spatially. Both FPRaker and the baseline are configured to have scaled-up PE array and the baseline are configured to have scaled-up PE array. Table III reports the corresponding area and power per tile. This is only possible if we can afford to use more FPRaker units than conventional units. One approach is to use an iso-compute area constraint. That is to determine how many FPRaker tiles we can fit in the same area for a baseline tile. For this we take into account only the compute cores as associated logic and not the scratchpads.

The conventional PE we compared against process concurrently 8 pairs of bfloat16 values and accumulates their sum. We can include buffers for the inputs (A and B) and the outputs so that we can exploit data reuse temporally. We can also organize multiple PEs in a grid sharing buffers and inputs across rows and columns to also exploit reuse spatially. Both FPRaker and the baseline are configured to have scaled-up GPU Tensor-Core-like tiles that perform $8 \times 8$ vector-matrix multiplication where 64 PEs are organized in a $8 \times 8$ grid and each PE performs 8 MAC operations in parallel.

Post layout, and taking into account only the compute area, an FPRaker tile occupies 22% the area vs. the baseline tile. Table III reports the corresponding area and power per tile. Accordingly, to perform an iso-compute-area comparison, we configure the baseline accelerator to have 8 tiles and FPRaker with 36 tiles. The area for the on-chip SRAM global buffer is $344\text{mm}^2$, $93.6\text{mm}^2$, and $334\text{mm}^2$ for the activations, weights, and gradients, respectively.

C. Execution Time

Figure 11 shows the performance breakdown of FPRaker due to the base-delta compression (BDC), and skipping zero and out-of-bound (OB) terms relative to the baseline. On average, FPRaker outperforms the baseline by $1.5 \times$ (skipping zero terms: +9%, BDC: +5.8%, skipping out-of-bound terms: +35.2%). From the studied convolution-based models, ResNet18-Q benefits the most from FPRaker where the performance improves by $2.04 \times$ over the baseline. Training for this network incorporates PACT quantization and as a result most of the activations and weights throughout the training process can fit in 4b or less. This translates into high term sparsity which FPRaker exploits. This result demonstrates that FPRaker can deliver benefits with specialized quantization methods without requiring that the hardware be also specialized for this purpose.

SNLI, NCF, and Bert are dominated by fully connected layers, while in fully-connected layers there is no weight reuse among different output activations, training can take advantage of batching to maximize weight reuse across multiple inputs (e.g., words) of the same input sentence which results in higher utilization of the tile PEs. Speedups follow bit sparsity. For example, FPRaker achieves a speedup of $1.8 \times$ over the baseline for SNLI due its high bit sparsity.

D. Energy Efficiency

Figure 11 shows the total energy efficiency of FPRaker over the baseline architecture for each of the studied models. On average, FPRaker is $1.4 \times$ more energy efficient compared to the baseline considering only the core logic and $1.36 \times$ more energy efficient when everything is taken into account. The energy-efficiency improvements follow closely the performance benefits. For example, benefits are higher at around $1.7 \times$ for SNLI and Detectron2. The quantization in ResNet18-Q boosts the core logic energy efficiency to as high as $1.97 \times$. Fig. 12 shows the energy consumed by FPRaker normalized to the baseline as a breakdown across three main components: core logic, off-chip and on-chip data transfers. Fig. 12 further breaks down FPRaker core into “Compute” (PE stages 1 and 2), “Control” (PE control units and shared term encoders), and “Accumulation” (PE stage 3). FPRaker along with the exponent base-delta compression reduce the energy consumption of the core logic and off-chip memory significantly.

E. Performance Analysis

Skipped Terms: Figure 13 shows a breakdown of the terms FPRaker skips. There are two cases: 1) skipping zero terms, and 2) skipping non-zero terms that are out-of-bounds due to the limited precision of the floating-point representation. Skipping out-of-bounds terms increases term sparsity for ResNet50-S2 and Detectron2 by around 10% and 5.1%, respectively. Networks with high sparsity (zero values) such as VGG16 and SNLI benefit the least from skipping out-of-bounds terms with the majority of term sparsity coming from zero terms. This is because there are few terms to start with. For ResNet18-Q, most benefits come from skipping zero terms as the activations and weights are effectively quantized to 4b values. However, as Figure 11 showed, skipping out-of-bound terms improves performance much more than the fraction of terms skipped would suggest. Recall, that all lanes must wait for the slowest one to finish processing amplifying the effect on performance across all lanes. In the worse case, all other 7 lanes are waiting for a single one to finish, wasting 7 execute slow. Skipping out-of-bound terms reduces this synchronization overhead.

Computation Phase: Figure 14 reports speedup for each of the 3 phases of training: the $A \times W$ in forward propagation, and the $A \times G$ and the $G \times W$ to calculate the weight and input gradients in the backpropagation, respectively. FPRaker consistently outperforms the baseline for all three phases. The
speedup depends on the amount of term sparsity, and the value distribution of A, W, and G across models, layers, and training phases. The less terms a value has the higher the potential for FPRaker to improve performance. However, due to the limited shifting that the FPRaker PE can perform per cycle (up to 3 positions) how terms are distributed within a value impacts the number of cycles needed to process it. This behavior applies across lanes to the same PE and across PEs in the same tile. In general, the set of values that are processed concurrently will translate into a specific term sparsity pattern. FPRaker favors patterns where the terms are close to each other numerically. **Where Cycles Go:** Figure 15 reports a breakdown of PE lane utilization and highlights performance bottlenecks. There are several reasons why stalls occur: 1) inter-PE synchronization, 2) intra-PE synchronization, and 3) Sharing the exponent block. Intra-PE synchronization stalls can happen in two scenarios: a) imbalance in the number of terms assigned for each lane within a PE due to uneven distribution of the term sparsity in the model which results in idle lanes waiting for the slowest lane to finish execution (“no term”), b) high span across consecutive terms within a lane which cause stalls due to the limited per cycle shift range of the PE (“shift range”). Stalls due to sharing the exponent block are rare. The more bit sparsity a model has the higher pressure on the exponent block and the higher the chance that it will not be able to keep up. Exponent stalls are noticeable for ResNet18-Q since the values there are effectively 4b. We can see a similar behavior for SNLI due to its high bit sparsity. However, there are other types of stalls that reduce pressure on the shared exponent block. First are stalls due to the limited per cycle shifting ability of the PEs. These are relatively few thus demonstrating that this technique presents a good performance vs. area trade-off. Second are stalls due to cross-lane term imbalance. These are the highest cause of PE underutilization; 32.8% on average, and at most 55% for NCF. Term imbalance is lowered if we reduce the number of lanes per PE or if we add weight buffers to allow faster lanes to proceed with the next set of weights albeit with a higher area overhead. However, doing so would increase the cost of the PE. This investigation is left for future work. Third are stalls due to inter-PE synchronization which are also rare. The ability for PE columns to run ahead by just one set sufficiently hides these stalls. Figure 16 shows the benefit of skipping out-of-bound terms in reducing the synchronization overheads (an average of 30.3% overall reduction) by improving the load balancing across the PE lanes. **Performance over Time:** Figure 18 shows the speedup of FPRaker over the baseline over time and throughout the training process for all the studied networks. The measurements show three different trends. For VGG16 speedup is higher for the first 30 epochs after which it declines by around 15% and plateaus. For ResNet18-Q, the speedup increases after epoch 30 by around 12.5% and stabilizes. This can be attributed to the PACT clipping hyperparameter being optimized to quantize activations and weights within 4-bits or below. For the rest of the networks, speeds ups remain stable throughout the training process. Overall, the measurements show that performance of FPRaker is robust and that it delivers performance improvements across all training epochs. **Effect of Tile Organization:** As shown in Figure 19, increasing the number of rows per tile reduces performance by 6% on average. This reduction in performance is due to synchronization among a larger number of PEs per column. When the number of rows increases, more PEs share the same set of A values. An A value that has more terms than the others will now affect a larger number of PE which will have to wait to finish processing. Since each PE processes a different combination of input vectors, each can be affected differently by intra-PE stalls such as “no term” stalls or “limited shifting” stalls. Figure 20 shows a breakdown of where time goes in each configuration. It can be seen that the stalls for the inter-PE synchronization increase and so do those for stalling for other lanes (“no term”). **Accuracy Study**

To study the effect of training with FPRaker on accuracy, we emulated the bit-serial processing of FPRaker PE during end-to-end training in PlaidML [76] which is a machine learning framework based on an OpenCL compiler at the backend. We force PlaidML to use the mad() function for every multiply-add during training. We override the mad() function with our implementation to emulate the processing of the FPRaker PE. We trained ResNet18 on CIFAR-10 and CIFAR-100 datasets as shown in Fig. 17. The blue line shows the top-1 validation accuracy for training natively in PlaidML with FP32 precision. The baseline performs bit-parallel MAC with I/O operands precision in bfloat16 which is known to converge and supported in the industry, e.g. in Google’s TPU. The figure shows that
both the baseline and FPRaker emulated versions converge at epoch 60 for both datasets with accuracy difference within 0.1% relative to the native training version. This is expected since FPRaker skips only ineffectual work, i.e., work which does not affect final result in the baseline MAC processing.

Synchronization Overhead Breakdown

- Top-1 Validation Accuracy
  - SqueezeNet 1.1
  - VGG16
  - ResNet50-S2
  - ResNet18-Q
  - SNLI
  - Image2Text
  - Detectron2
  - NCF
  - Bert
  - Geomean

- Speedup Breakdown
  - AxG
  - GxW
  - AxW

- Lane Efficiency
  - useful
  - shift range
  - exponent

- Synchronization Overhead Breakdown
  - no term
  - inter-PE

G. Per Layer Accumulator Width Profiling

Conventionally, training uses bfloat16 for all computations. As we noted in the introduction, there have been proposal for using mixed datatype arithmetic where some of the computations used fixed-point instead [55], [58], [59], [60]. Sakr et. al, propose to use floating-point where however the number of bits used by the mantissa varies per operation and per layer [61]. We use the suggested mantissa precisions while training AlexNet and ResNet18 on Imagenet. Figure 21 shows the performance of FPRaker following this approach. FPRaker can dynamically take advantage of the variable accumulator width per layer to skip the ineffectual terms mapping outside the accumulator boosting overall performance. Training ResNet18 on ImageNet with per layer profiled accumulator width boosts the speedup of FPRaker by 1.51×, 1.45× and 1.22× for A × W, G × W and A × G, respectively achieving an overall speedup of 1.56× over the baseline compared to 1.13× that is possible when training with a fixed accumulator width. Adjusting the mantissa length while using a bfloat16 container manifests itself a suffix of zero bits in the mantissa.

VI. RELATED WORK

We have discussed a broad spectrum of software level training acceleration techniques and some accelerator designs in the introduction. Since FPRaker is a processing element level method it can in principle be used with any of these techniques to further boost performance and energy efficiency. However, there will be interactions which would need to be investigated. For example, since FPRaker requires more parallelism to match and exceed the throughput of a conventional bit-parallel PE it will interact with data reuse/dataflow selection methods. However, data parallelism is abundant during training due to the use of batching. The investigation of these interactions while certainly interesting is left for future work. We believe that the experiments presented in this study sufficiently demonstrate that FPRaker is a technique worthwhile considering further.

We have already commented on the conceptual differences between bit-Pragmatic and FPRaker in the introduction which also apply to Laconic, which is another design that performs bit skipping for inference with fixed-point values. Further, we reported that converting bit-Pragmatic to process floating-point values results in a design that is less energy efficient compared to an optimized bit-parallel unit. We also converted Laconic to floating-point and unfortunately the results were equally disappointing. The design is not small enough nor energy efficient enough to improve over optimized bit-parallel floating point units.

Compared to inference, training performs more computation and requires a larger volume of data to be kept around. While in inference activations and weights are used only once during training they are used twice and more importantly the dataflow is different between these two uses. This challenges several optimizations that are otherwise possible in inference. For example, inference accelerators targeting sparsity such as
all contributing to a single final value. The processing element was developed as a building block for accelerators for training neural networks. We were motivated by the relatively high term level sparsity that all value exhibit during training. While we evaluated FPRaker for training, it can naturally also be used for inference. While many neural network models can use fixed-point arithmetic there are models that still require floating-point. For example, these include models that process language or recommendation systems. Of course, whether FPRaker will provide benefits for inference with such models needs to be demonstrated.

FPRaker opens new avenues of research in efficient precision training. Different precision can be assigned to each layer during training depending on the layer’s sensitivity to quantization. Further, training can start with lower precision and increase the precision per epoch near conversion. FPRaker can adapt dynamically to different precisions and rewards innovations in training algorithms by boosting performance and energy efficiency. Whether the benefits we demonstrate are compelling enough for practitioners to commit to deploying FPRaker in the next generation designs can be of course be the subject of debate. Regardless, demonstrating this novel and practical approach to acceleration for floating-point values is of significant value to architects. We are confident that FPRaker will spur additional work targeting computation and memory hierarchy optimizations that exploit term sparsity for improving training and inference.

REFERENCES

[1] 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

[2] Z. Yang, Z. Dai, Y. Yang, J. Carbonell, R. Salakhutdinov, and Q. V. Le, “Xlnet: Generalized autoregressive pretraining for language understanding,” 2019.

[3] R. Zellers, A. Holtzman, H. Rashkin, Y. Bisk, A. Farhadi, F. Roesner, and Y. Choi, “Defending against neural fake news,” in Advances in Neural Information Processing Systems 32, H. Wallach, H. Larochelle, A. Beygelzimer, F. dAlché-Buc, E. Fox, and R. Garnett, Eds. Curran Associates, Inc., 2019, pp. 9054–9065. [Online]. Available: http://papers.nips.cc/paper/9106-defending-against-neural-fake-news.pdf

[4] C. J. Shallue, J. Lee, J. Antognini, J. Sohl-Dickstein, R. Frostig, and G. E. Dahl, “Measuring the effects of data parallelism on neural network
F. Seide, H. Fu, J. Droppo, G. Li, and D. Yu, "PipeDream: Generalized pipeline parallelism for dnn training," in Proceedings of the 27th ACM Symposium on Operating Systems Principles, ser. SOSP ’19, New York, NY, USA: ACM, 2019, pp. 1–15. [Online]. Available: http://doi.acm.org/10.1145/3343130.3359646

Y. Huang, Y. Cheng, A. Bapna, O. Firat, D. Chen, M. Chen, H. Lee, J. Ndjim, Q. V. Le, Y. Wu et al., "Gpipe: Efficient training of giant neural networks using pipeline parallelism," in Advances in Neural Information Processing Systems, 2019, pp. 103–112.

M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper, and B. Catanzaro, "Megatron-LM: Training multi-billion parameter language models using gpus model parallelism," arXiv preprint arXiv:1909.08053, 2019.

L. Song, J. Mao, Y. Zhuo, X. Qian, H. Li, and Y. Chen, "Hypar: Towards hybrid parallelism for deep learning accelerator array," 2019 IEEE International Symposium on High Performance Computer Architecture (HPCA), Feb 2019. [Online]. Available: http://dx.doi.org/10.1109/hpca.2019.00027

W. Wen, C. Xu, F. Yan, C. Wu, Y. Wang, Y. Chen, and H. Li, "Tergrd: Terminating gradients to reduce communication in distributed deep learning," in Proceedings of the 31st International Conference on Neural Information Processing Systems, ser. NIPS’17. USA: Curran Associates Inc., 2017, pp. 1508–1518. [Online]. Available: http://dl.acm.org/citation.cfm?id=3294771.3294915

K. Hegde, J. Yu, R. Agrawal, M. Yan, M. Pellauer, and C. W. Fletcher, "Ucm: Exploiting computational reuse in deep neural networks via weight checkpointing," in Proceedings of the 45th Annual 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

S. H. Hashemi, S. A. Jouhi, and R. H. Campbell, “Tictac: Accelerating distributed deep learning with communication scheduling,” arXiv preprint arXiv:1803.03288, 2018.

Y. Peng, Y. Zhu, Y. Chen, Y. Bao, B. Yi, C. Lan, C. Wu, and C. Guo, "A generic communication scheduler for distributed dnn training acceleration," in Proceedings of the 27th ACM Symposium on Operating Systems Principles, ser. SOSP ’19. New York, NY, USA: ACM, 2019, pp. 16–29. [Online]. Available: http://doi.acm.org/10.1145/3343130.3359642

A. Jayarajan, "Priority-based parameter propagation for distributed deep neural network training," Ph.D. dissertation, University of British Columbia, 2019.

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.

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 Int’l Symp. on Computer Architecture, 2016.

J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. Enright-Jerger, and A. Moshovos, "CNVLUTIN: Ineffectual-Neuron-Free Deep Neural Network Computing," in Int’l Symp. on Computer Architecture, 2016.

D. Kim, J. Ahn, and S. Yoo, “Zeno: Zero-aware neural network accelerator,” IEEE Design Test, vol. 35, pp. 39–46, 2018.

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 Int’l Conf. on Computer Architecture (ISCA), 2016.

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 Int’l Symp. on Computer Architecture, ser. ISCA ’17, 2017.

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 Int’l Symp. on Microarchitecture, 2016.

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 Int’l Symp. on Microarchitecture, 2018.
G. Henry, P. T. P. Tang, and A. Heinecke, “Leveraging the bfloat16 artificial intelligence datatype for higher-precision computations,” 2019 IEEE 26th Symposium on Computer Arithmetic (ARITH), Jun 2019. [Online]. Available: https://doi.org/10.1109/ARITH.2019.00019

A. Heinecke, P. Dubey, J. Corbal, N. Shustrov, R. Dubtsov, E. Fomenko, and V. O. Pirogov, “Mixed precision training of convolutional neural networks,” in Proceedings of the 46th International Symposium on Computer Architecture, 2019, pp. 304–317. [Online]. Available: https://doi.org/10.1145/3326985.3326995

NVIDIA, “Training with mixed precision,” https://docs.nvidia.com/deeplearning/sdk/mixed-precision-training/index.html.

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. ACM, 2018, p. 20.

J. Lee, C. Kim, S. Kang, D. Shin, S. Kim, and H. Yoo, “UNPU: an energy-efficient deep neural network accelerator with fully variable weight bit precision,” J. Solid-State Circuits, vol. 54, no. 1, pp. 173–185, 2019. [Online]. Available: https://doi.org/10.1109/JSSC.2018.2865489

A. D. Lascorz, S. Sharify, I. E. Vivancos, D. M. Stuart, O. M. Awad, P. Judd, M. Mahmoud, M. Nikolic, K. Siu, Z. Poulos, and A. Moshovos, “Shapeshifter: Enabling fine-grain data width adaptation in deep learning,” in Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2019, Columbus, OH, USA, October 12-16, 2019. ACM, 2019, pp. 28–41. [Online]. Available: https://doi.org/10.1145/3352460.3358289

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. IEEE Computer Society, 2018, pp. 764–765.

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

S. Sharify, A. D. Lascorz, M. Mahmoud, M. Nikolic, K. Siu, D. M. Stuart, Z. Poulos, and A. Moshovos, “Lacoon deep learning inference acceleration,” in Proceedings of the 46th International Symposium on Computer Architecture, ISCA 2019, Phoenix, AZ, USA, June 22-26, 2019, pp. 304–317. [Online]. Available: https://doi.org/10.1145/3307650.3322255

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

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

C. Sakr, N. Wang, C.-Y. Chen, J. Choi, A. Agrawal, N. Shanbhag, and K. Gopalakrishnan, “Accumulation bit-width scaling for ultra-low precision training of deep neural networks,” 2019.

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.

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.

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

Z. Wang and J.-C. Liu, “Translating math formula images to latex sequences using deep neural networks with sequence-level training,” 2019.

T.-Y. Lin, M. Maire, S. Belongie, J. Bourdev, D. Girshick, S. Guadarrama, and L. Fei-Fei, “Microsoft coco: Common objects in context,” 2014.

F. M. Harper and J. A. Konstan, “The movielens datasets: History and context,” ACM Trans. Interit. Syst., vol. 5, no. 4, Dec. 2015. [Online]. Available: https://doi.org/10.1145/2827872

D. Elliott, S. Frank, K. Sima’an, and L. Specia, “Multi30k: Multilingual english-german image descriptions,” 2016.

C. Sakr, N. Wang, C.-Y. Chen, J. Choi, A. Agrawal, N. Shanbhag, and K. Gopalakrishnan, “Accumulation bit-width scaling for ultra-low precision training of deep networks,” 2019.

G. Pekhimenko, V. Seshadri, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, “Base-delay/intermediate compression: Practical data compression for on-chip caches,” in Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques, ser. PACT ’12. New York, NY, USA: Association for Computing Machinery, 2012, p. 377–388. [Online]. Available: https://doi.org/10.1145/2370816.2370870

P. Kurup and T. Abbasi, Logic Synthesis Using Synopsys, 2nd ed. Springer Publishing Company, Incorporated, 2011.

Cadence, “Innovus implementation system,” https://www.cadence.com/cadence-www/global/en_US/home/tools/digital-design-and-sig/hierarchical-design-and-floorplanning/innovus-implementation-system.html.

S. Galal and M. Horowitz, “Energy-efficient floating-point unit design,” IEEE Trans. Computers, vol. 60, no. 7, pp. 913–922, 2011. [Online]. Available: https://doi.org/10.1109/TC.2010.121

Hewlett Packard, “Cacti,” https://github.com/HewlettPackard/cacti.

I. Micron Technology, “Dd4r power calculator 4.0,” https://www.micron.com/-media/documents/products/power-calculator/dd4r_power_calc.xslm.

I.A. PlaidML, “PlaidML readthedocs-hosted.com/en/latest/index.html. [Online]. Available: https://vertaesi-plaidml.readthedocs-hosted.com/en/latest/index.html
[77] J. R. Shinde and S. S. Salankar, “VLSI implementation of bit serial architecture based multiplier in floating point arithmetic,” in 2015 International Conference on Advances in Computing, Communications and Informatics, ICACCI 2015, Kochi, India, August 10-13, 2015. IEEE, 2015, pp. 1672–1677. [Online]. Available: https://doi.org/10.1109/ICACCI.2015.7275854

[78] P. Chau, K. Chew, and W. Ki, “A bit-serial floating-point complex multiplier-accumulator for fault-tolerant digital signal processing arrays,” in IEEE International Conference on Acoustics, Speech, and Signal Processing. IEEE, 1987, pp. 483–486.

[79] B. Feinberg, U. K. R. Vengalam, N. Whitehair, S. Wang, and E. Ipek, “Enabling scientific computing on memristive accelerators,” in Proceedings of the 45th Annual International Symposium on Computer Architecture, ser. ISCA ’18. IEEE Press, 2018, p. 367–382. [Online]. Available: https://doi.org/10.1109/ISCA.2018.00039

[80] T. Xiao, J. Zhang, K. Yang, Y. Peng, and Z. Zhang, “Error-driven incremental learning in deep convolutional neural network for large-scale image classification,” in Proceedings of the ACM International Conference on Multimedia, MM ’14, Orlando, FL, USA, November 03 - 07, 2014, 2014, pp. 177–186. [Online]. Available: https://doi.org/10.1145/2647868.2654926