Mixing Low-Precision Formats in Multiply-Accumulate Units for DNN Training

Mariko Tatsumi, Silviu-Ioan Filip, Caroline White, Olivier Sentieys, Guy Lemieux

Mixing Low-Precision Formats in Multiply-Accumulate Units for DNN Training

Mariko Tatsumi*, Silviu-Ioan Filip†, Caroline White*, Olivier Sentieys† and Guy Lemieux*
*Department of Electrical and Computer Engineering, University of British Columbia, Vancouver, Canada
†Univ. Rennes, Inria, Rennes, France

***Abstract*—The most compute-intensive stage of deep neural network (DNN) training is matrix multiplication where the multiply-accumulate (MAC) operator is key. To reduce training costs, we consider using low-precision arithmetic for MAC operations. While low-precision training has been investigated in prior work, the focus has been on reducing the number of bits in weights or activations without compromising accuracy. In contrast, the focus in this paper is on implementation details beyond weight or activation width that affect area and accuracy. In particular, we investigate the impact of fixed- versus floating-point representations, multiplier rounding, and floating-point exceptional value support. Results suggest that (1) low-precision floating-point is more area-effective than fixed-point for multiplication, (2) standard IEEE-754 rules for subnormals, NaNs, and intermediate rounding serve little to no value in terms of accuracy but contribute significantly to area, (3) low-precision MACs require an adaptive loss-scaling step during training to compensate for limited representation range, and (4) fixed-point is more area-effective for accumulation, but the cost of format conversion and downstream logic can swamp the savings. Finally, we note that future work should investigate accumulation structures beyond the MAC level to achieve further gains.**

I. Introduction

With the increasing adoption of deep learning (DL), particularly with embedded/edge computing scenarios and the increasing complexity of state-of-the-art DNNs, there is great interest in improving the cost/performance ratio of DNN training. To make training more affordable, one has to inevitably use low-precision arithmetic (e.g. 8-bit and lower formats), as opposed to the current de facto standard of 32-bit IEEE-754 single-precision floating-point (FP32). Smaller data sizes can reduce the hardware needed and can potentially improve memory footprint, bandwidth, and power.

While low-precision DNN training has been investigated by prior art, few works directly examine hardware implementation complexity. Accordingly, the precise hardware resources saved by replacing FP32 with a low-precision data type is left ambiguous. However, reducing the resource demand per MAC unit, while maintaining accuracy, is crucial to deploying more computational power within a fixed hardware budget.

In order to address these ambiguities we apply different low-precision data formats to the MAC units used within a general matrix multiply (GEMM) accelerator for DNN training and investigate the impact on training accuracy and hardware resource utilization in an FPGA context.

In some previous work, the exact methodology used for implementing low-precision calculations is not described. To do so accurately, one would need to avoid standard accelerated libraries and emulate each low-precision calculation, requiring greater effort to code and leading to a significant slowdown of the training process. One quick technique, used for instance by QPyTorch [1], applies a quantization layer after each composite computational layer like convolution or GEMM. In such a scheme, all intermediate calculations use a higher precision, and only the final results are 'downconverted' to the reduced precision. Such techniques do not properly model precision loss, leading to overly optimistic results with inflated training accuracy. In this paper, we accurately model the arithmetic of each individual operator and storage unit within a custom-written GEMM kernel that has two implementations (FPGA and GPU) which are cross-validated for bit-level accuracy.

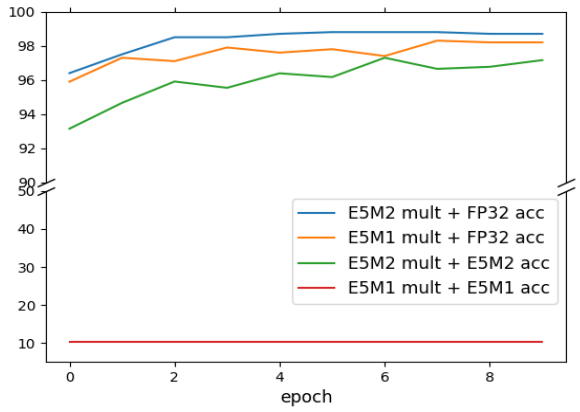

Fig. 1. Validation accuracy observed using LeNet5+MNIST.

Consider, for example, the validation accuracy of training LeNet5+MNIST shown in Fig 1, where all multiplier inputs are quantized to 5 exponent bits and 1 or 2 mantissa bits (E5M1 and E5M2, respectively). With FP32 accumulation followed by quantization (blue/orange curves), both E5Mm formats look viable, but with low-precision accumulation in the E5M2 format (green curve) accuracy is slightly degraded. Worse yet, the E5M1 format (red curve) result is catastrophic and never converges. This shows the importance of accurately modeling all stages of low-precision arithmetic.

The main contributions/observations are as follows:

- a new, extensible, open-source DNN training framework called Archimedes-MPO which uses FPGA or GPU

acceleration to study the resource-accuracy tradeoffs of using custom data types and operators;
- relaxing IEEE-754 rules (i.e., removing subnormal support and repurposing exceptional value encodings) for multiplication saves area and maintains accuracy;
- individual operator area is minimized using different number representations (i.e., a floating-point multiplier and a fixed-point accumulator), but format conversion and additional downstream logic (due to the wider output) quickly overcomes any apparent savings; and
- in a system like ours (where no effort is made by hardware or software to limit the number of terms that appear in an accumulation chain), the size of a fixed-point accumulator is more sensitive to the target model/dataset than a floating-point multiplier/accumulator.

II. Background and Related Work

We first review DNN training together with a technique called loss scaling that is used in low-precision training to maintain representation range and accuracy.

A. DNN Training

DNNs are a sequence of layers which usually take the form

$$f(\mathbf{x}) = g(\mathbf{W}^T\mathbf{x} + \mathbf{b}) \tag{1}$$

where $\mathbf{x} \in \mathbb{R}^m$ is a layer input vector, $f(\mathbf{x}) \in \mathbb{R}^n$ is the layer output, $\mathbf{W} \in \mathbb{R}^{m\times n}$ and $\mathbf{b} \in \mathbb{R}^n$ are trainable parameters (weight and bias terms), and g is a non-linear activation function applied element-wise. The application of Eq. (1) on all layers in the network is called inference.

Equation (1) is at the core of most layers, convolutions included. A common approach is to cast a two-dimensional convolution operation into a matrix multiplication operation coupled with so-called `im2col` and `col2im` transforms [2].

DNN training consists of finding a set of parameters that (approximately) minimizes an empirical loss function

$$\mathcal{L}(\boldsymbol{\mathcal{W}}) = \frac{1}{B}\sum_{i=1}^{B}\ell_i(\boldsymbol{\mathcal{W}}) \tag{2}$$

where $\boldsymbol{\mathcal{W}}$ is the complete set of trainable parameters in the network, each $\ell_i(\boldsymbol{\mathcal{W}})$ is an averaged loss function on a batch of N input data samples, and B is the number of training batches. To simplify notation, we have avoided specifying the dependence of loss functions on input data.

A form of stochastic gradient descent (SGD) is generally used to minimize $\mathcal{L}$. At each iteration t, SGD will randomly pick a batch from the training set (i.e., a loss ℓ from $\{\ell_i\}_{i=1}^{B}$) and update the parameters according to

$$\boldsymbol{\mathcal{W}}^{(t+1)} = \boldsymbol{\mathcal{W}}^{(t)} - \alpha_t\nabla\ell(\boldsymbol{\mathcal{W}}^{(t)}) \tag{3}$$

with $\alpha_t \in \mathbb{R}_+$ the current learning rate and $\nabla\ell(\boldsymbol{\mathcal{W}}^{(t)})$ the parameter gradient. This update is applied to every parameter in each layer of the network by using the chain rule. The process is known as back-propagation, with the loss gradient with respect to the output of a layer k called the activation gradient (or delta), which we denote by $\boldsymbol{\delta}_k$. If $z(\mathbf{x}) = \mathbf{W}^T\mathbf{x}$, back-propagation through a layer k requires computing the forward path using Eq. (1) and the backward path for the delta and weight gradient respectively:

$$\boldsymbol{\delta}_{k-1} = \frac{\partial z}{\partial \mathbf{x}}\frac{\partial g}{\partial z}\boldsymbol{\delta}_k \text{ and } \nabla\ell(\mathbf{W}) = \frac{\partial z}{\partial \mathbf{W}}\frac{\partial g}{\partial z}\boldsymbol{\delta}_k. \tag{4}$$

Since $\partial z/\partial\mathbf{x} = \mathbf{W}$ and $\partial z/\partial\mathbf{W} = \mathbf{x}$, both equations in Eq. (4) reduce to a matrix multiplication operation with $\frac{\partial g}{\partial z}\boldsymbol{\delta}_k$. In total, each iteration calculates Eq. (3) of batch DNN training with $N > 1$ batch size usually requires three GEMM operations per layer: two for computing the gradients as just mentioned, and one for computing the layer inputs, which require storing the intermediate results evaluated during the forward path.

B. Loss Scaling

One of the potential problems of using low-precision during training is the limited dynamic range. This is especially important when computing gradient values, which decrease in magnitude as training progresses. Their values might thus fall below the minimum values that low-precision data types can represent, potentially leaving most of their representation range unused. A possible solution [3] is to scale the loss function by a large value S before the start of the backward pass so that the new (scaled) gradients do not underflow in the low-precision format. The parameter update in Eq. (3) is then performed with the gradient divided by S.

Adaptive Loss Scaling: One difficulty with loss scaling is picking a good scaling factor. Choosing an appropriate value is essential in certain cases, but searching for the best value can be time consuming. The approach proposed in [4] dynamically adjusts the factor as follows: start with a large value of S (e.g. 1024) and if no overflow is detected within a certain number of iterations P, the value of S is doubled; if overflow is detected, the current iteration is skipped and S is halved. The impact on runtime is negligible.

C. Related Work

As DNNs have increased in size, reducing precision below FP32 has been studied for both training and inference.

The current trend in low-precision training with hardware support focuses on 16-bit floating-point formats with half-precision (FP16) [3] and bfloat16 [5] alternatives from NVIDIA and Google. More aggressive options using 8-bit [6]–[9] and even 4-bit [10] formats have also been shown to achieve FP32-level results on specific training scenarios. The NVIDIA Hopper GPU architecture offers support for two different FP8 formats [11]. Since submission, other work has confirmed that training with FP8 formats coupled with scaling factors [12] or various exponent bias values [13] can match 32-bit level baselines. Accumulation in these approaches is usually done in 16-bit or 32-bit floating-point arithmetic.

Other approaches have used fixed-point types during training [14]–[19]. For instance, [14] uses INT8/16 for storage and INT32 for accumulation to yield FP32-comparable results. Later work [15] used INT16 accumulation during GEMM

operations, but prediction accuracy degraded, whereas [17] dynamically changed the storage format from INT4 to INT8 during training to achieve FP32-comparable results.

The efficient implementation of quantized DNNs is also widely studied in the FPGA community [20]–[24]. In particular, [21] and [24] combined floating-point multipliers and fixed-point accumulators. They determined that this combination has the potential to achieve a good prediction accuracy-resource utilization tradeoff for inference tasks; in present work, we consider using a similar combination for DNN training tasks. On the training side, [22] successfully trained VGG16 and PreResNet-20 networks [25] on an embedded FPGA using the SWALP [26] algorithm by replacing the GEMM multipliers with 8-bit block floating-point operators but retaining 32-bit integer accumulators; the impact of reducing accumulator size is not investigated.

On the simulation side, QPyTorch [1] extends the PyTorch library by adding rounding modules before and after each computation module. This approximates low-precision arithmetic by converting high-precision results into smaller formats. It allows quick testing of ideas, but internal operations within a composite operation (e.g. GEMM) are still done in FP32, making it inaccurate because it neglects overflows and rounding that must occur in low-precision hardware.

Another relevant framework is Barista [27], which extends the Caffe library by adding FPGA acceleration support. One advantage of our work over Barista is that we are using training results measured on real FPGA hardware, whereas Barista is limited to a performance model.

There is also a large body of work on quantization aware training methods (e.g., the Brevitas [28] library for FPGAs), but such methods generally modify the training process to compress models and accelerate inference, not training.

III. Design and Implementation

To explore the MAC design space, we developed Archimedes-MPO, a DNN training framework with mixed-precision operations. It supports CPUs, GPUs, and FPGAs.

The FPGA-enabled version, shown in Fig. 2, uses PetaLinux, TinyDNN, and an HLS-based GEMM accelerator [29]. FPGA support does provide acceleration, but its main purpose is to validate training accuracy and implementation area of low-precision MAC operations within an accelerated system. As a result, it lacks many system-level optimizations expected by end applications.

GPU simulation support, which is bit-accurate with the FPGA version, allows training to scale out so we can explore multiple training scenarios concurrently.

In both GPU and FPGA versions, we implement custom precision GEMM kernels since these are the core operations in both convolution and fully connected layers. All other network components (e.g., activation functions and batch normalization), are performed in software on the host CPU using FP32. We have limited our exploration to GEMMs for expedience and because they amount to the most computation and are the most likely to exhibit precision problems during training.

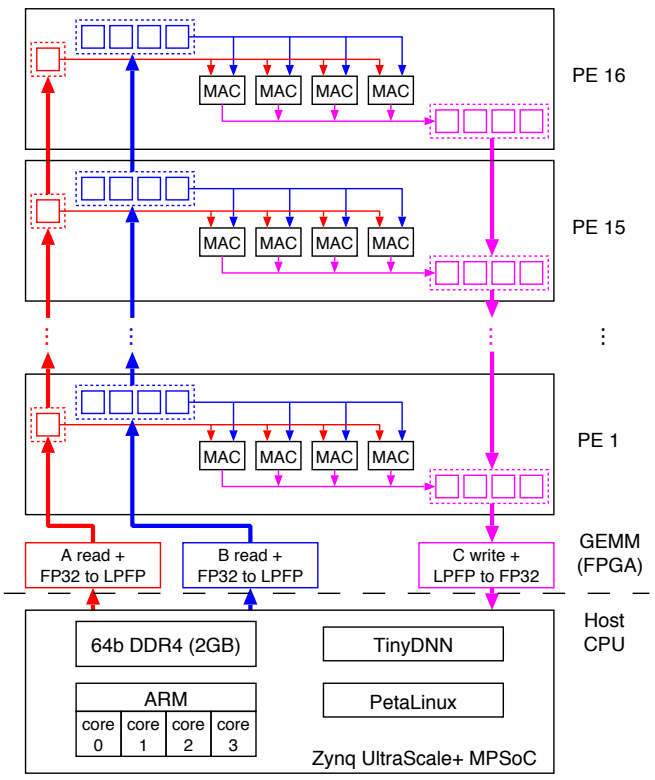

Fig. 2. Archimedes-MPO Block Diagram for FPGAs.

A more in-depth study that also customizes the arithmetic of other operations during training, notably the parameter update in Eq. (3), is left for future work.

A. Host Implementation

Archimedes-MPO is a hybrid software/hardware approach that builds upon the TinyDNN [30] framework. Written in C++14, TinyDNN is a pure software, header-only, dependency-free, templated and optimized DL library capable of multi-threaded SIMD execution on x86 and ARM.

Archimedes-MPO modifies TinyDNN in three key ways: (a) it adds more data types (i.e., fixed-point and floating-point with custom precisions) to extend each layer type with C++ templates and operator overloading; (b) it employs custom low-precision GPU or FPGA versions for the GEMM kernel; and (c) it implements `col2im` and `im2col` transforms needed to offload convolutional layers using the GEMM kernel.

With templates and overloading, Archimedes-MPO can create networks with a different precision and data type for each layer instance. Some layers can specify different precisions for different operations, e.g. multiplier inputs, outputs, and accumulation in GEMM. However, all layer instances that use the FPGA-accelerated GEMM hardware must use the same formats because only one hardware kernel is synthesized.

We explore the use of custom data types within the GEMM kernel and compute all other layers on the host using FP32. Conversion between FP32 and the custom types is done in the

accelerated GEMM kernels; this improves compute efficiency, but does not improve memory footprint or bandwidth.

By leaving network operations on the host in FP32, our trained models have FP32 weight values, and that parameter updates are also done in FP32. Reducing the precision during these updates is subject of other research, e.g., [31]. Nevertheless, our GEMM calculations do use low precision.

B. GPU Implementation

Archimedes-MPO on GPU enables fast batch-mode training on x86/GPU servers running Linux. Since it uses the same TinyDNN core as the FPGA version, it is easy to track changes between the two (e.g. if we modify training strategies). The GEMM kernel uses CUDA to emulate low-precision arithmetic in a way that is bit-accurate with the FPGA accelerator. To decrease the overhead caused by data transfer between layers, all operations (other than the image data loading at the start of each iteration) are implemented as CUDA kernels.

C. FPGA Implementation

The FPGA version of Archimedes-MPO uses a heavily customized version of an open-source GEMM hardware kernel [29] written in C using Xilinx Vitis HLS. The Vitis framework also relies upon PetaLinux and the Vitis Embedded Platform (an FPGA logic shell).

The original GEMM-HLS kernel [29] supports only standard-width primitive data types (i.e., FP32 and `uint8`). We modified it to support small custom data types, such as FP8 with different exponent-mantissa breakdowns. In addition, we modified data reading/writing modules in the GEMM kernel to convert data between FP32 and low-precision during transfer. While TinyDNN can use any precision in each layer, FP32 is used on the host side to keep things simple, reduce the overhead of emulating other precisions, and avoid unnecessary data type conversions. During conversion, values outside the representation range are cast to $\pm\infty$. Similarly, multiplier and accumulator operators propagate $\pm\infty$ when necessary.

We also modified the GEMM-HLS kernel to directly read both transposed ($n \times m$) and non-transposed ($m \times n$) matrix data. This does not affect the resulting clock speed, and kernel throughput remains over 97% of peak GOPS when tested with sufficiently large (2048×2048) transposed and non-transposed matrices. This allowed us to remove the redundant input data transpose during back-propagation. Finally, we also modified GEMM-HLS to remove the use of DSP blocks everywhere outside of the PEs, such as address generation logic.

The FPGA kernels are implemented by targeting a Xilinx ZCU104 development board. It contains a Zynq UltraScale+ MPSoC device (ZCU7EV) and one 2GB bank of 64-bit DDR4 memory. The GEMM kernel is synthesized into the FPGA logic using Vitis HLS version 2020.2. The software portion of our framework utilizes all four Arm Cortex-A53 cores running at 1.2GHz. Compiler auto-vectorization is enabled to use NEON instructions. The FPGA GEMM kernel uses a linear array of 16 PEs, with each PE containing 4 MAC units, for a total of 64 parallel MAC units running at 280MHz. In this system, each MAC sums an entire dot product and the number of addends depends upon the network model.

TABLE I
FLOATING- AND FIXED-POINT MULTIPLIER AREA

input & output precision	LUTs	DSPs
FP32 (no DSP)	987	0
FP32	374	2
FP16 / bfloat16	195 / 180	1
E6M3 (`CFG1`)	115	0
E5M3 (`CFG1`)	86	0
E6M2 to E5M1	(see Fig 3)	0
Q16.16	279	4
Q8.8	106	1
Q7.7	93	1
Q6.6	81	1

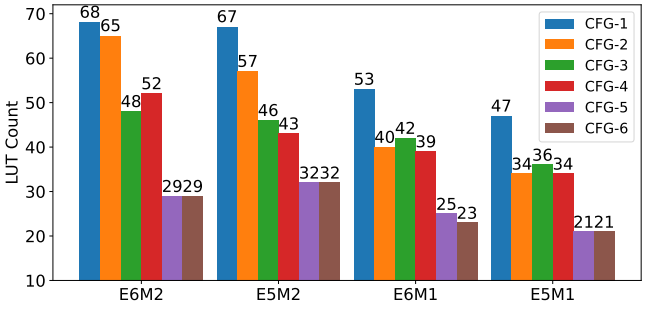

Fig. 3. Area of floating-point multiplier variants.

1) Multiplier: Conventionally, a floating-point product is computed by adding the exponents and multiplying the mantissas of two operands. With larger mantissa fields, the mantissa product is normally computed using the FPGA DSP blocks. Since we are working with small mantissa fields of 3 or fewer bits, we have instead used look-up tables (LUTs). We call this base implementation `CFG1`. Given that FPGAs typically contain 6-input LUTs, a single LUT can compute a single output bit of a 3×3 product. LUTs are far smaller than DSPs, which are normally intended for 16-bit or larger mantissas. For example, in a Xilinx UltraScale+ VUP13 FPGA, for every DSP block there are 140 6-input LUTs [24]; these LUTs can implement 23 independent 3×3 integer multipliers.

We also implemented a fixed-point multiplier following standard conventions: the multiplier calculates the product of two integer values and rounds the result back to the input data type. This implementation uses DSP blocks because the required fixed-point formats are much wider.

Resource Utilization: Table I shows the resources used by a single multiplier using various formats. Floating-point formats are reported as EeMm, where e and m indicate the exponent and mantissa sizes. Fixed-point formats use Q$i.f$ notation, where i and f are the integer and fractional part sizes. The precise formats reported were chosen because they achieved FP32-comparable results in our early experiments (not reported here). Note that Table I results include support for subnormals, NaNs, infinity and round-to-nearest.

To see the reduction in resource utilization, Table I also provides baseline values for 16- and 32-bit multipliers. The

TABLE II
ALTERNATIVE ENCODING SCHEMES FOR E5M2

	NaN/∞		Subnormal		
mantissa	conventional CFG1 to CFG3	custom CFG4 to CFG6	conventional CFG1 to CFG4	custom CFG5	truncate CFG6
b00	∞	65,536	0	0	0
b01	NaN	81,920	1.53E-5	3.81E-5	0
b10	NaN	98,304	3.05E-5	4.58E-5	0
b11	NaN	∞	4.58E-5	5.34E-5	0

low-precision floating-point multipliers save considerable resources and notably use zero DSP blocks. The mantissa size more strongly affects area than the exponent length (e.g. E5M2 vs. E5M1 and E5M2 vs. E6M2 in Fig. 3). This matches expectations since extra mantissa bits in multipliers typically require more logic than extra exponent bits.

The fixed-point multipliers tend to use more LUTs than EeMm multipliers, plus they all need DSP blocks. Given that EeMm needs only 6 to 10 total bits while Q$i.f$ needs 12 to 32 bits to represent values, Q$i.f$ multipliers may also need more system support for wider buffering and interconnect than EeMm. Given that low-precision floating-point multipliers also have better range and good training results (shown later), fixed-point multiplication seems untenable.

2) Multiplier Variants: In this subsection, we enumerate several possible multiplier implementations. Their impact on DNN training is evaluated in Section IV-B2. Each variant below includes the changes of all previous variants.

CFG1 - Default: The first multiplier variant discussed in the previous subsection has ancillary logic to handle exceptional values, namely subnormal inputs and outputs, one output rounding mode, and NaN values.

CFG2 - Removal of Subnormal Outputs: The second multiplier variant removes support for subnormal output values. It saves several LUTs, yet it leads to information loss as subnormal outputs are all truncated to 0.

CFG3 - Removal of Output Rounding: The third multiplier variant produces an *exact* product by removing output rounding and widening the output sufficiently to avoid all precision loss. The product of two input values, each with m_i+1 mantissa bits after including the implicit bit, generates $2m_i+2$ output bits. Reducing this to fewer bits requires rounding and renormalizing, which incurs extra logic overhead. Removal of output rounding simultaneously reduces LUT count for multiplier and improves accuracy by avoiding all information loss. Note that output saturation to $\pm\infty$ remains, and inputs to the subsequent adder also change to $2m_i+2$ bits.

CFG4 - Removal of NaNs: This variant removes NaN output values because they waste some encoding range. By treating NaNs as normalized values, we can extend the encoding range without additional logic. This requires remapping $\pm\infty$ to use a mantissa field of all ones, which is also more logical and consistent. Table II shows the expanded range.

CFG5 and CFG6 - Alternative Subnormal Inputs: The fifth and sixth multiplier variants attempt to save more LUTs by changing the way subnormal inputs are treated. Although this also affects the size of data conversion logic from FP32, the focus is on the size of MAC units which are far more numerous. Multiplication of denormalized input values requires dedicated logic, so CFG5 treats this part of the encoding range as normalized values to obviate the dedicated logic. Meanwhile, CFG6 truncates all subnormal input values to 0 so they become trivial.

Table II shows the custom encoding used for E5M2 subnormal input values. A CFG5 multiplier can still represent the upper portion of conventional subnormal numbers, but with extra precision. However, it loses all subnormal values with smaller exponent values. In contrast, CFG6 loses all subnormal values and is more likely to negatively impact training.

Resource Utilization: Fig. 3 shows the LUT counts used to implement the 6 multiplier variations across multiple data types: E6M2, E5M2, E6M1, and E5M1. For all data types, the LUT count of a single multiplier decreases from CFG1 to CFG6 as expected, and DSP count is always 0.

The first thing we notice is a considerable LUT count reduction when removing the logic for handling subnormal output values (i.e., CFG1 vs. CFG2).

Similarly, the reduction that arises from the removal of output rounding is shown when comparing CFG2 to CFG3. Interestingly, when the multiplier benefits a lot from the removal of subnormal output handling logic, for example saving more than 10 LUTs, its LUT count does not differ significantly between CFG2 and CFG3 (e.g. E6M1, and E5M1). On the other hand, when the reduction amount by the first modification is not remarkable, the multipliers tend to benefit more from removing the output rounding logic (e.g. E6M2). In all cases, multipliers having the same number of mantissa bits settle down to a similar LUT count after removing both subnormal outputs and output rounding. We believe this variation is caused by the sometimes inconsistent results of HLS synthesis optimizations.

In contrast, removing the extra shifters required by subnormal input values from CFG4 to CFG5 or CFG6 consistently decreases the LUT count by 10 to 13. Overall, both CFG5 and CFG6 are more than 50% smaller than CFG1, **demonstrating a higher area sensitivity to multiplier policy than size in bits**. Since CFG5 has a larger range with the same area, it is superior to CFG6.

3) Accumulator: Similar to the multiplier investigation, we studied accumulator resource use with low-precision floating-point and fixed-point data types. Both floating-point/fixed-point accumulators are custom implemented following a standard approach for adders. Fixed-point accumulation includes saturation logic. Floating-point accumulation includes subnormals, swapping and shifting of operands plus extra bits (implied, guard, round, and sticky bits), hence using an $m+4$-bit adder with m-bit mantissas. The baseline FP32 accumulator uses the Xilinx LogiCORE IP Library; it does not support subnormals and implements shifters in DSP blocks.

Resource Utilization: The Table III Accumulator columns show the resources used for different data types. For each floating-point accumulator, we also show the expected input type for the associated floating-point multiplier (CFG5).

TABLE III
AREA OF ACCUMULATOR AND DATA CONVERTER

FP-mult input (CFG5)	FP-mult output	Accumulator LUTs	Accumulator DSPs	Converter (to Q8.13) LUTs
FP32	FP32	189 (no subnormal support)	2	-
E6M3	E7M7	255	0	116
E6M2	E7M5	185	0	103
E6M1	E7M3	187	0	72
E5M3	E6M7	242	0	97
E5M2	E6M5	187	0	81
E5M1	E6M3	165	0	67
-	Q16.16	89	0	-
-	Q8.13	55	0	-
-	Q8.8	43	0	-
-	Q7.7	35	0	-
-	Q6.6	32	0	-

The fixed-point accumulators use considerably fewer LUTs than the floating-point ones because they do not require input/output alignment shifters or rounding.

4) Full MAC Unit: Based on the results so far, using a low-precision floating-point multiplier with a fixed-point accumulator uses the lowest LUT count for each operator. However, a MAC unit built this way also requires a float-to-fixed converter which uses costly data shifters.

The Table III Converter column shows the LUT count for converting the specified multiplier output precision to the Q8.13 fixed-point format; the Q8.13 precision was selected based upon training accuracy results in Section IV-B4.

The converters are larger than the Q8.13 accumulator itself. Using a total of 122 to 171 LUTs, they are closer to floating-point accumulators in size. Hence, the implementation cost of type conversion cannot be ignored.

IV. TRAINING RESULTS

To explore the accuracy impact of different low-precision GEMM configurations we trained ResNet-20 [32] and VGG16 [33] networks on the CIFAR-10 [34] dataset. On a slightly larger scale, we also train a ResNet-50 model using a subset of ImageNet. We use the GPU version for rapid experimentation; while the FPGA version works, GPUs are more readily available to run thousands of training experiments.

A. Training Settings

We have used SGD with a momentum value of 0.9 for all experiments. The initial learning rate and weight decay are set to 0.1/0.01 and 0.0001/0.0005 for ResNet-20 and VGG16 training, respectively, following the experimental settings of the original papers. All experiments employ a cosine annealing scheduler to decay the learning rate as the training proceeds. The batch size is set to 128, and we have trained the ResNet-20 model for 165 epochs and the VGG16 model for 200 epochs; both epoch counts are calculated based on the training iteration count reported in the original papers. ResNet-50 training uses an initial learning rate of 0.01 and runs for 100 epochs with a batch size of 64. All experiments also use an adaptive loss scaling technique (see Sec. II-B), with initial scaling factor $S = 1024$ and update frequency $P = 200$. For image preprocessing, the 8-bit pixel values are converted to FP32 by dividing by 255 and normalized using the mean and standard deviation of the dataset. When the first layer is lower precision, the input values are quantized accordingly.

B. Training Results

At first, we evaluate top-1 test accuracy after independently replacing either the multiplier or accumulator with lower precision. After that, we investigate accuracy of replacing the multiplier with one low-precision choice, and the accumulator with a different choice. The goal is to remain within 1 percentage point of FP32 results, which we call *FP32-comparable*.

1) Adaptive Loss Scaling: Results for using adaptive loss scaling with low-precision multiplication are shown in Fig. 4(a). Using more than 5 exponent bits were all FP32-comparable (not shown), whereas E5Mm and E4Mm suffered from accuracy degradation (dashed lines) due to the limited representation range. By applying adaptive loss scaling (solid lines), gradients are shifted into the representable range and FP32-comparable results are restored for all except E4M1 and E4M2 multipliers with ResNet-20. Although not shown, low-precision accumulators showed similar sensitivity where loss scaling boosted accuracy to FP32-comparable results for E5Mm with 4/5 or more mantissa bits, and for E4Mm with 5/6 or more mantissa bits with VGG16/ResNet-20, respectively. Also not shown, non-adaptive loss scaling produced lower quality of results. We deem adapative loss scaling to be essential, so we adopt it for all subsequent experiments.

2) Multiplier Investigation: Fig. 4(b) shows test accuracy of E4Mm multiplier variants for different mantissa sizes. Removing subnormal output in CFG2 degrades accuracy across all mantissa sizes; this loss is restored with CFG3 by widening the output and removing rounding. Next, CFG4 reuses the NaN encoding space and improves accuracy slightly. On VGG16, both CFG5 and CFG6 are FP32-comparable, but on ResNet-20 CFG6 degrades unacceptably while CFG5 degrades only 2%.[1,2] Below, we will consider E4M2 with CFG5, since it is almost FP32-comparable, as well as E5M2.

3) Accumulator Investigation: Fig. 4(c) shows the test accuracy using different floating-point accumulators, while Fig. 4(d) uses fixed-point accumulators.

FP32-comparable training results are achieved with E7Mm/E6Mm/E5Mm accumulators using more than 5 mantissa bits, as well as E4Mm accumulators using more than 4/5 mantissa bits on ResNet-20/VGG16, respectively. As mentioned earlier, adaptive loss scaling was instrumental at improving results of E5Mm and E4Mm with larger m, but with smaller m problems persist, likely due to truncation error, also known as *swamping* [7], [36], when adding small values to a larger value. Compensated summation [36] can mitigate swamping, but widening the mantissa is easier.

With fixed-point accumulation, both ResNet-20 and VGG16 models can reduce the accumulator size to Q8.12 while maintaining FP32-comparable results.

[1]As explained in [35], we adopt $S = 4096$ with CFG5/6 multipliers.
[2]Although not shown, E5m is FP32-comparable for CFG5 and CFG6.

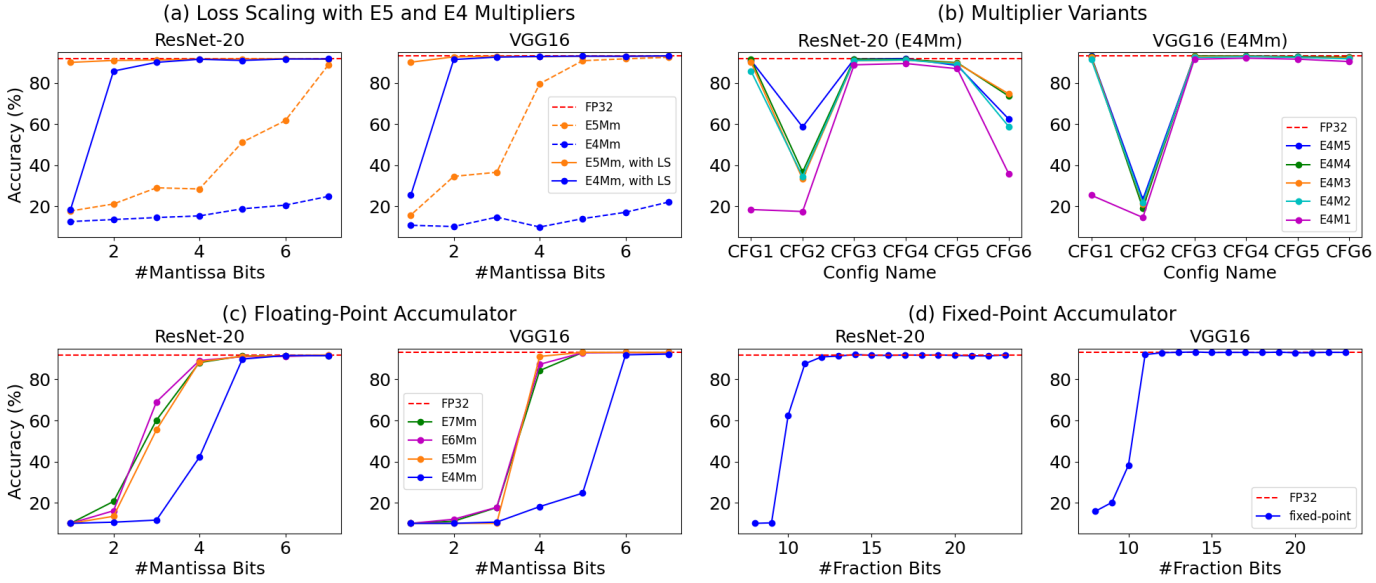

Fig. 4. The graphs show the best top-1 test accuracy achieved by training with different configurations. The dashed red line always shows the best accuracy baseline achieved with FP32 training: (a) E5Mm (orange) and E4Mm (blue) multiplier accuracy before (dashed lines) and after (solid lines) adaptive loss scaling; (b) accuracy of E4Mm multiplier variants, where the blue/green/orange/cyan/magenta lines represent m=5/4/3/2/1 precisions; (c) accuracy of floating-point accumulators, where green/magenta/orange/blue lines represent e=7/6/5/4 exponent bits and the x-axis indicates the m mantissa bits; and (d) accuracy of fixed-point accumulators, where the x-axis indicates the number of fractional bits that follow 8 integer bits.

TABLE IV
SYSTEM-LEVEL AREA AND TEST ACCURACY

configurations	LUTs	DSPs	ResNet-20/ CIFAR-10 Acc. (%)	ResNet-50/ Imagewoof Acc. (%)
FP32	58,420	320	91.85	57.57
E5M2(CFG5) + E6M5	42,640	0	91.04	59.79
E5M2(CFG5) + Q8.13	43,471	0	90.95	10.92
E5M2(CFG5) + Q11.13	44,876	0	not tested	59.38

4) MAC Investigation: Fig. 5 shows test accuracy when both multiplier and accumulator use lower precision. Observations from Sec. IV-B2 and Sec. IV-B3 lead us to start with an E4M2 (CFG5) multiplier and either E5M5 or Q8.12 accumulators. E5M5 accumulation converges but accuracy is 3.7% lower than FP32. Fixed-point accumulation fails to converge. Given the accuracy degradation in both cases, we first increased the exponent width of the multiplier to E5M2. This restores FP32-comparable accuracy, but fixed-point accumulation still fails to converge. Increasing the accumulator to Q8.13 achieves FP-32 comparable accuracy.

Table IV summarizes the system-level total LUT count and accuracy. Mixed-precision training requires around 25% fewer LUTs and no DSPs compared to the FP32 baseline, while limiting accuracy loss to within 1 percentage point.

Even after accounting for data conversion, we did not expect the fixed-point accumulator configuration to have a higher system-level LUT count. The increase was due to an increase in downstream interconnect and buffering logic from the wider output of the fixed-point accumulator (21 bits versus 12 bits for floating-point). To save downstream logic, we attempted to truncate the final accumulation value into a smaller fixed-point representation, but accuracy quickly degraded. There is an opportunity to explore other accumulation techniques, e.g. [37], at the MAC and system level.

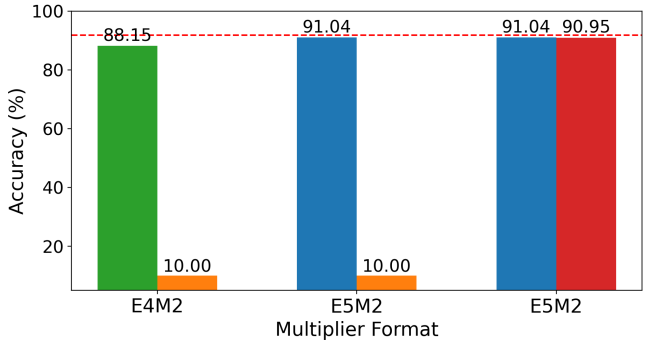

Fig. 5. Best CIFAR-10 test accuracy achieved with ResNet-20 training. The green/blue bars show the accuracy of using E5M5/E6M5 accumulators, and orange/red bars show the accuracy of using Q8.12/Q8.13 accumulators.

C. Extensibility to Larger Training Tasks

To test further, we used a subset of ImageNet called Imagewoof [38] with a ResNet-50 model. Imagewoof reduces training effort compared to ImageNet by 99%, but keeps recognition accuracy challenging, by using just 10 dog classes out of the total 1,000 classes. Results in Table IV indicate the E6M5 accumulator configuration achieves FP32-comparable results, but the fixed-point accumulator requires 3 more bits (Q11.13) to converge.

From this, we suspect the size of a fixed-point accumulator is more sensitive to the model and dataset than a floating-point accumulator or multiplier; the lack of an exponent field

requires a much wider word length. Future work on system-level techniques may reduce accumulator size and limit this sensitivity, e.g. chunk-based accumulation [7], or fixed-to-float conversion of accumulated results.

V. Limitations

The study is limited to only a few CNN models. We use smaller CNNs because our initial focus is on edge-based training, but larger CNNs are needed to validate the results. (We also trained MLPs on CIFAR10 and MNIST, but excluded those results for brevity.) However, the intent is to go beyond CNNs and explore other network types, such as transformers, and use-cases, such as transfer learning.

This work does not use larger datasets (e.g. full ImageNet) due to limitations with our training infrastructure. Being bit-accurate is very slow; we estimate a single ImageNet training instance to require over 60 days. Unfortunately our GPU systems impose time quota of 14 days which cannot be extended. To circumvent this limitation, we are adding checkpointing into our TinyDNN framework.

In addition, this work does not consider data formats other than floating-point and fixed-point formats. Other research has shown good potential for block floating-point [26], [39]–[42], posits [43]–[45], logarithmic number systems [46]–[49] and even binary formats [50]. The investigation of mixed data formats DNN training including those data types, especially for block mini-floats [41], is left for future work.

In addition, other than loss scaling, this work has not investigated significant changes to the training algorithms which may affect range and precision needs. This would likely improve results further. Although the work quantifies that FP8 saves bits and area, the resulting energy and memory footprint savings are left to future work as well.

VI. Conclusions

We have investigated the use of mixed-precision operations at the MAC level of a GEMM kernel during DNN training from a resource-accuracy tradeoff perspective. To do so, we have developed a framework, Archimedes-MPO, that allows the use of custom fixed and floating-point arithmetic at each network layer via custom GEMM kernels for GPU and FPGA. We then explored low-precision sensitivities on image classification tasks with the CIFAR-10 dataset on ResNet-20 and VGG16 CNNs, and then performed a final evaluation on ResNet-50 using a subset of ImageNet. Archimedes-MPO is availble under an open-source license.

Our results agree with prior work that 8-bit floating-point formats can be used to successfully train some CNNs, and that adaptive loss scaling is essential. However, we go beyond the bit width investigations done in previous work. For example, we provide evidence that avoiding certain IEEE-754 conventions for multiplication (e.g., output rounding, subnormals, NaN and ∞ encodings) at the MAC level can lead to **both lower area and better accuracy**.

In addition, when optimizing the $\times$ and $+$ operators independently, we find that floating-point is more resource-efficient for multiplication, and its area is more sensitive to multiplier policy than size. In particular, we note that 8-bit floating-point formats use very little area because two 3-bit mantissas can be multiplied using 6-input LUTs, i.e., no DSPs are needed. In contrast, fixed-point is more efficient for accumulation, but its size is sensitive to the network model and dataset.

When combined at the MAC and system level, the area of data conversion and growth of downstream logic become a concern for fixed-point accumulation.

Despite the limited models and datasets used, we believe our overall conclusions to be sound: (1) multipliers should use a narrow floating-point format, (2) significant multiplier area can be saved, yet precision maintained, by eliminating output rounding and reducing exceptional value support, (3) for small mantissas, multiplier area is more sensitive to these policies than the precise mantissa size, and (4) fixed-point accumulators are very attractive (small size, no precision loss from rounding), but large structures are needed for format conversion (from the floating-point multiplier output) and system-level downstream logic (e.g. interconnect and buffering).

Although we did attempt to minimize the number of bits for low-precision training, we do not believe our values to be the 'final answer' due to previously stated limitations. To answer this, future work must investigate using a variety of network models, other training datasets, and changes to the training process, e.g., limiting accumulation chain lengths.

In our experiments, we demonstrated that the area used by the multiplier can be kept small if exceptional value support is modified and the number of mantissa bits is kept small. Still, it may be worthwhile to investigate other changes to training algorithms and number systems, such as multiplierless training where logarithmic numbers are used [46], [51].

There are many directions for future work. Accumulation architecture at the MAC and system levels can be improved. For example, future work should consider breaking long accumulation chains, either in software or hardware, to limit the number of required accumulator bits. To save area, individual MAC adders and final accumulator outputs should be narrow while addressing the challenges of swamping, range and final test accuracy. Since GEMM hardware kernels can differ in how, where and when addition is done, there is a rich design space at the hardware level; this can be further augmented with software tools which break NN models into smaller chunks.

This work would also benefit from an error analysis to help fine-tune format requirements and further support our practical findings in a more principled manner. Beyond using adaptive loss scaling, assessing the resource-accuracy impact of other training computations (e.g. parameter update, activation function evaluation) is also important and should help guide the design of training algorithms and processes that are tailored for a low-precision regime.

Finally, we intend to expand upon our study by using more models (beyond CNNs) and datasets.

REFERENCES

[1] T. Zhang, Z. Lin, G. Yang, and C. De Sa, "QPyTorch: A Low-Precision Arithmetic Simulation Framework," *https://arxiv.org/abs/1910.04540*, 2019.

[2] Y. Jia, "Learning Semantic Image Representations at a Large Scale," Ph.D. dissertation, University of California at Berkeley, 2014.

[3] P. Micikevicius *et al.*, "Mixed Precision Training," in *ICLR*, 2018.

[4] NVIDIA, "Train with Mixed Precision," Tech. Rep., 2022. [Online]. Available: https://docs.nvidia.com/deeplearning/performance/pdf/Training-Mixed-Precision-User-Guide.pdf

[5] D. D. Kalamkar *et al.*, "A Study of BFLOAT16 for Deep Learning Training," 2019. [Online]. Available: http://arxiv.org/abs/1905.12322

[6] X. Sun *et al.*, "Hybrid 8-bit Floating Point (HFP8) Training and Inference for Deep Neural Networks," in *NeurIPS*, 2019.

[7] N. Wang, J. Choi, D. Brand, C.-Y. Chen, and K. Gopalakrishnan, "Training Deep Neural Networks with 8-Bit Floating Point Numbers," in *NeurIPS*, 2018, p. 7686–7695.

[8] C. Sakr, N. Wang, C. Chen, J. Choi, A. Agrawal, N. R. Shanbhag, and K. Gopalakrishnan, "Accumulation Bit-Width Scaling For Ultra-Low Precision Training Of Deep Networks," in *ICLR*, 2019.

[9] N. Mellempudi, S. Srinivasan, D. Das, and B. Kaul, "Mixed Precision Training With 8-bit Floating Point," 2019. [Online]. Available: http://arxiv.org/abs/1905.12334

[10] X. Sun *et al.*, "Ultra-Low Precision 4-bit Training of Deep Neural Networks," in *NeurIPS*, 2020, pp. 1796–1807.

[11] NVIDIA, "NVIDIA H100 Tensor Core GPU Architecture," Tech. Rep., 2022. [Online]. Available: https://resources.nvidia.com/en-us-tensor-core/gtc22-whitepaper-hopper

[12] P. Micikevicius *et al.*, "FP8 Formats for Deep Learning," 2022. [Online]. Available: https://arxiv.org/abs/2209.05433

[13] B. Noune *et al.*, "8-bit Numerical Formats for Deep Neural Networks," 2022. [Online]. Available: https://arxiv.org/abs/2206.02915

[14] R. Banner, I. Hubara, E. Hoffer, and D. Soudry, "Scalable Methods for 8-Bit Training of Neural Networks," in *NeurIPS*, 2018, p. 5151–5159.

[15] Y. Yang, L. Deng, S. Wu, T. Yan, Y. Xie, and G. Li, "Training High-Performance and Large-Scale Deep Neural Networks with Full 8-bit Integers," *Neural Networks*, vol. 125, pp. 70–82, 2020.

[16] C. De Sa, M. Leszczynski, J. Zhang, A. Marzoev, C. R. Aberger, K. Olukotun, and C. Ré, "High-Accuracy Low-Precision Training," 2018. [Online]. Available: http://arxiv.org/abs/1803.03383

[17] Y. Fu *et al.*, "FracTrain: Fractionally Squeezing Bit Savings Both Temporally and Spatially for Efficient DNN Training," in *NeurIPS*, 2020.

[18] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, "Deep Learning with Limited Numerical Precision," in *ICML*, 2015.

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

[20] C. Luo, Y. Wang, W. Cao, P. H. Leong, and L. Wang, "RNA: An Accurate Residual Network Accelerator for Quantized and Reconstructed Deep Neural Networks," in *Int'l Conf. on Field-Program. Logic*, 2018.

[21] C. Wu, M. Wang, X. Chu, K. Wang, and L. He, "Low Precision Floating Point Arithmetic for High Performance FPGA-based CNN Acceleration," in *Int'l Symp. on FPGAs*, 2020.

[22] S. Fox, J. Faraone, D. Boland, K. Vissers, and P. H. Leong, "Training Deep Neural Networks in Low-Precision with High Accuracy Using FPGAs," in *FPT*, 2019.

[23] Y. Cao, C. Wang, and Y. Tang, "Explore Efficient LUT-based Architecture for Quantized Convolutional Neural Networks on FPGA," in *Int'l Symp. on Field-Program. Cust. Computing Mach.*, 2020, pp. 232–232.

[24] Xilinx, "Higher Performance Neural Networks with Small Floating Point," Tech. Rep. WP530, 2021.

[25] K. He, X. Zhang, S. Ren, and J. Sun, "Identity Mappings in Deep Residual Networks," in *Euro. Conf. on Computer Vision*, 2016.

[26] G. Yang, T. Zhang, P. Kirichenko, J. Bai, A. G. Wilson, and C. De Sa, "SWALP: Stochastic Weight Averaging in Low Precision Training," in *ICML*, 2019, pp. 7015–7024.

[27] D. A. Vink, A. Rajagopal, S. I. Venieris, and C.-S. Bouganis, "Caffe Barista: Brewing Caffe with FPGAs in the Training Loop," in *Int'l Conf. on Field-Program. Logic*, 2020, pp. 317–322.

[28] A. Pappalardo, "Xilinx/Brevitas," 2021. [Online]. Available: https://doi.org/10.5281/zenodo.3333552

[29] J. de Fine Licht, G. Kwasniewski, and T. Hoefler, "Flexible Communication Avoiding Matrix Multiplication on FPGA with High-Level Synthesis," in *Int'l Symp. on FPGAs*, 2020, pp. 244–254.

[30] T. Noumi, *Tiny-DNN: Header only, Dependency-free Deep Learning Framework in C++14*, 2017. [Online]. Available: http://tiny-dnn.readthedocs.io

[31] P. Zamirai, J. Zhang, C. R. Aberger, and C. De Sa, "Revisiting BFloat16 Training," *http://arxiv.org/abs/2010.06192*, 2020.

[32] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," in *CVPR*, 2016, pp. 770–778.

[33] K. Simonyan and A. Zisserman, "Very Deep Convolutional Networks for Large-Scale Image Recognition," in *ICLR*, 2015.

[34] A. Krizhevsky, "Learning Multiple Layers of Features from Tiny Images," University of Toronto, Tech. Rep., 2009.

[35] M. Tatsumi, "Using Mixed Low-Precision Formats in Multiply-Accumulate (MAC) Units for DNN Training," 2022. [Online]. Available: https://open.library.ubc.ca/collections/ubctheses/24/items/1.0412995

[36] N. J. Higham, "The Accuracy of Floating Point Summation," *SIAM J. Scientific Computing*, vol. 14, no. 4, pp. 783–799, 1993.

[37] F. de Dinechin *et al.*, "An FPGA-Specific Approach to Floating-Point Accumulation and Sum-of-Products," in *FPT*, 2008.

[38] J. Howard, "Imagenette, Imagewoof and Imagewang." [Online]. Available: https://github.com/fastai/imagenette/

[39] D. Das *et al.*, "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*. OpenReview.net, 2018. [Online]. Available: https://openreview.net/forum?id=H135uzZ0-

[40] U. Köster *et al.*, "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. Red Hook, NY, USA: Curran Associates Inc., 2017, p. 1740–1750.

[41] S. Fox, S. Rasoulinezhad, J. Faraone, D. Boland, and P. Leong, "A Block Minifloat Representation for Training Deep Neural Networks," in *International Conference on Learning Representations*, 2021. [Online]. Available: https://openreview.net/forum?id=6zaTwpNSsQ2

[42] M. Drumond, T. LIN, M. Jaggi, and B. Falsafi, "Training DNNs with Hybrid Block Floating Point," in *Advances in Neural Information Processing Systems*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, Eds., vol. 31. Curran Associates, Inc., 2018. [Online]. Available: https://proceedings.neurips.cc/paper/2018/file/6a9aeddfc689c1d0e3b9ccc3ab651bc5-Paper.pdf

[43] J. L. Gustafson and I. T. Yonemoto, "Beating Floating Point at its Own Game: Posit Arithmetic," *Supercomput. Front. Innov.*, vol. 4, no. 2, pp. 71–86, 2017. [Online]. Available: https://doi.org/10.14529/jsfi170206

[44] R. Chaurasiya, J. Gustafson, R. Shrestha, J. Neudorfer, S. Nambiar, K. Niyogi, F. Merchant, and R. Leupers, "Parameterized Posit Arithmetic Hardware Generator," in *2018 IEEE 36th International Conference on Computer Design (ICCD)*, 2018, pp. 334–341.

[45] Y. Uguen, L. Forget, and F. de Dinechin, "Evaluating the Hardware Cost of the Posit Number System," in *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*, 2019, pp. 106–113.

[46] J. Zhao, S. Dai, R. Venkatesan, M.-Y. Liu, B. Khailany, B. Dally, and A. Anandkumar, "Low-Precision Training in Logarithmic Number System using Multiplicative Weight Update," 2021. [Online]. Available: https://arxiv.org/abs/2106.13914

[47] M. S. Ansari, B. F. Cockburn, and J. Han, "An Improved Logarithmic Multiplier for Energy-Efficient Neural Computing," *IEEE Transactions on Computers*, vol. 70, no. 4, pp. 614–625, 2021.

[48] D. Miyashita, E. H. Lee, and B. Murmann, "Convolutional Neural Networks using Logarithmic Data Representation," 2016. [Online]. Available: https://arxiv.org/abs/1603.01025

[49] E. H. Lee, D. Miyashita, E. Chai, B. Murmann, and S. S. Wong, "LogNet: Energy-efficient neural networks using logarithmic computation," in *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2017, pp. 5900–5904.

[50] E. Wang, J. J. Davis, D. Moro, P. Zielinski, J. J. Lim, C. Coelho, S. Chatterjee, P. Y. Cheung, and G. A. Constantinides, "Enabling Binary Neural Network Training on the Edge," in *Proceedings of the 5th International Workshop on Embedded and Mobile Deep Learning*, 2021, pp. 37–38.

[51] A. R. Nair, P. K. Nath, S. Chakrabartty, and C. S. Thakur, "Multiplierless MP-Kernel Machine For Energy-efficient Edge Devices," 2021. [Online]. Available: https://arxiv.org/abs/2106.01958