Dynamically Adapting Floating-Point Precision to Accelerate Deep Neural Network Training

John Osorio Ríos*, Adrià Armejach*, Eric Petit‡, Greg Henry‡ and Marc Casas*
*Barcelona Supercomputing Center (BSC) - Universitat Politècnica de Catalunya (UPC), Barcelona, Spain
†Intel, Oregon, USA
Email: john.osorio@bsc.es

***Abstract*—Mixed-precision (MP) arithmetic combining both single- and half-precision operands has been successfully applied to train deep neural networks. Despite its advantages in terms of reducing the need for key resources like memory bandwidth or register file size, it has a limited capacity for diminishing further computing costs, as it requires 32-bits to represent its output. On the other hand, full half-precision arithmetic fails to deliver state-of-the-art training accuracy. We design a binary tool SERP based on Intel Pin which allows us to characterize and analyze computer arithmetic usage in machine learning frameworks (Pytorch, Caffe, Tensorflow) and to emulate different floating point formats. Based on empirical observations about precision needs on representative deep neural networks, this paper proposes a seamless approach to dynamically adapt floating point arithmetic. Our dynamically adaptive methodology enables the use of full half-precision arithmetic for up to 96.4% of the computations when training state-of-the-art neural networks; while delivering comparable accuracy to 32-bit floating point arithmetic. Microarchitectural simulations indicate that our *Dynamic* approach accelerates training deep convolutional and recurrent networks with respect to FP32 by 1.39× and 1.26×, respectively.**

***Index Terms*—Reduced Precision, bfloat16, Mixed Precision (MP), Binary Analysis Tool, Dynamic Precision.**

I. INTRODUCTION

The use of Deep Neural Networks (DNNs) is becoming ubiquitous in areas like computer vision [1] or language translation [2]. DNNs display remarkable pattern detection capabilities. In particular, Convolutional Neural Networks (CNNs) are able to accurately detect and classify objects over large image data sets [1], and Recurrent Neural Networks (RNNs) using encoder-decoder models are capable of solving tasks like Neural Machine Translation (NMT) [3]. However, to achieve the desired levels of accuracy, a large amount of samples needs to be exposed to the models for tens or even hundreds of times during training. This fact drives training costs up in terms of resources like power, memory storage or compute time.

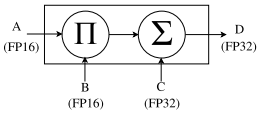
Several proposals successfully mitigate training costs by replacing the use of standard floating point 32-bit (FP32) arithmetic with alternative approaches that employ non-standard low precision data representation formats [4]–[6]; reducing memory storage, bandwidth requirements, and compute costs. Hardware vendors have incorporated half-precision data formats [7], [8] like the BFloat16 (BF16) format [8] and have implemented mixed-precision (MP) instructions, which aim at reducing memory bandwidth and storage consumption.

MP relies on *Fused Multiply-Add* (FMA) instructions that involve 16-bit inputs for the multiplier, and an FP32 accumulator that generate an FP32 output. Therefore, MP does not fully deliver the potential benefits of reduced precision arithmetic since it requires writing back to memory a 32-bit output, limiting the computational throughput of data-level parallelism techniques like vectorization; heavily used in mathematical and DNN libraries. MP constitutes an intermediate approach between the widely used 32-bit arithmetic and a full 16-bit approach. The latter can deliver larger performance improvements, but suffers from significant accuracy degradation when training state-of-the-art (SoA) neural networks.

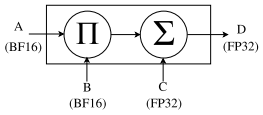
We design a binary instrumentation tool named SERP (Seamless Emulation of Reduced Precision Formats), which is based on Intel Pin and allows us to characterize and analyze computer arithmetic usage in machine learning frameworks (e.g., Pytorch, Caffe and Tensorflow), as well as to emulate any different floating point formats. We analyze multiple SoA CNN and an RNN models and find that over 98,15% of the total floating point instructions are FMAs. To better analyze network behavior from a computer arithmetic perspective, we observe the network's requirements in terms of floating point precision in various phases of the training algorithm and learning epochs.

Based on our empirical observations about precision needs in representative DNNs, this paper proposes a seamless approach that dynamically adapts floating point precision arithmetic. Our *Dynamic* approach enables the use of true half-precision arithmetic for most of the training process, achieving performance improvements and comparable training accuracy with respect to FP32 and MP. Our proposal employs simple heuristics based on the evolution of the training loss function to decide the adequate precision to use for a number of batches.

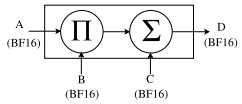
Our evaluation with SERP shows that the *Dynamic* approach is able to obtain comparable accuracy w.r.t FP32 and MP training for CNN and RNN models over the same number of iterations. For all evaluated CNNs, over 94.6% of the FMAs are performed entirely in half-precision (BF16), demonstrating that it is possible to use half-precision computations for a large portion of the training process without incurring in any accuracy degradation. In addition, we show that our heuristics are sensitive to the network requirements, increasing the amount of MP (or FP32) computations when half-precision is



(a) FMA FP16/FP32 inputs



(b) FMA BF16/FP32 inputs



(c) FMA all inputs BF16

Fig. 1: Alternatives when computing a *Fused Multiply-Add* (FMA) instruction.

TABLE I: SoA FMAs for training.

Training	Inputs		Output	Multiply	Accum.
	A,B	C	D		
Tensor cores	FP16/BF16	FP32	FP32	FP16/BF16	FP32
Google TPU v3	BF16	FP32	FP32	BF16	FP32
AVX512-BF16	BF16	FP32	FP32	FP32	FP32
Full BF16	BF16	BF16	BF16	BF16	BF16

not enough in terms of accuracy. Finally, we use the Sniper [9] architectural simulator to evaluate the performance benefits of our approach, since there is no real hardware supporting full BF16 FMA instructions. Our evaluation shows that the *Dynamic* approach accelerates training by 1.39× and 1.26× over FP32 for CNN and RNN, respectively; while keeping the same accuracy levels.

II. Background in Mixed-Precision Training and Motivation

MP diminishes training costs by reducing the data representation size of certain network components. Weights, activations and gradients are stored in half-precision. Importantly, some phases of the training process like computing weight updates (WU) and batch normalization (BN) layers require full FP32 precision, which requires representing the weights in 32-bits.

Nvidia GPUs support MP training by leveraging their tensor cores, which combine the floating point 16-bit (FP16) and FP32 [7] formats in FMA instructions. Figure 1a displays a MP FMA instruction, which computes $D = A \cdot B + C$. Input parameters A and B are represented in FP16 and the result is added to the third input C, typically a 32-bit weight. The final output D is also represented in FP32. This approach requires applying a scaling factor when converting FP32 values to FP16 to avoid range representation issues.

Similarly, hardware vendors [8], [10] propose combining the BF16 and FP32 formats in a single FMA instruction, which Figure 1b shows. Conversion from FP32 to BF16 does not require a scaling factor as both types have the same representation range and, therefore, the conversion just requires applying Round to Nearest Even (RNE) rounding.

MP FMA instructions bring significant benefits since they require less memory bandwidth and register storage than FP32 FMAs. However, using a full BF16 FMA like the one represented in Figure 1c provides even larger benefits. In terms of memory bandwidth, BF16 FMAs require moving 50% and 33% less data than FP32 and MP FMAs, respectively. Similarly, BF16 FMAs require one half and two thirds of FP32 and MP FMAs register storage, respectively.

Table I summarizes the input/output data types and the precision employed during the arithmetic operations in three SoA MP approaches. Both Nvidia GPUs (tensor cores) and Google TPUs [11] take multiplication inputs in half-precision format while the accumulator and accumulation arithmetic use FP32, producing an FP32 output. Recent Intel Xeon CPUs implement the new AVX512-BF16 extensions [12], which also feature half-precision multiply inputs but convert them to FP32 before the arithmetic operations, i.e., the FMA is entirely performed using FP32. Finally, we propose to use full BF16 FMA operations to train DNNs, which can provide significant savings in register storage, bandwidth, functional unit logic, and performance improvements by better leveraging data-level parallelism via vectorization.

To date, the implementation of full BF16 FMAs has not been adopted by hardware vendors, as training DNNs using BF16 FMAs does not provide SoA convergence properties, leading to lower accuracy with respect to MP and FP32 training. In this paper we describe a *Dynamic* approach that enables using BF16 FMAs for almost the entire training process while keeping the same convergence properties as FP32 and MP training. We apply all SoA FMA approaches to train four relevant models in Section III, and we analyze the reasons why a full 16-bit approach does not provide enough accuracy. Based on these observations, we describe our *Dynamic* precision approach in Section IV.

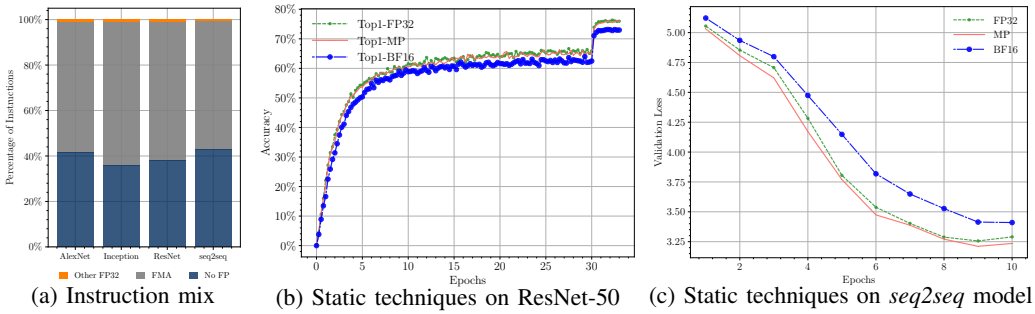
III. Analysis of State-of-the-Art Approaches

We analyze the different SoA approaches to perform FMA instructions on three CNN models: AlexNet [1], Inception V2 [13], [14], and ResNet-50 [15]; and an RNN model [3], [16] that solves the NMT task, referred in this work as sequence to sequence (*seq2seq*) model. Section VI contains the details regarding the configuration of each evaluated model and the methodology we follow.

A. Instruction Counts

Figure 2a shows the instruction mix for one batch on each network. We observe that floating point instructions constitute a large portion of these workloads. For example, they represent 58.44%, 60.93%, 62.95% and 56.44% of the total count for AlexNet, Inception V2, ResNet-50, and *seq2seq*, respectively. The amount of non-FMA FP instructions remains below 1.10% for the four networks. Therefore, FMA instructions constitute a large portion of the whole training workload. These measurements justify the focus on FMA instructions.

Prior research [7], [8] describes the need to use FP32 arithmetic in WU routines and BN layers when using training approaches based on MP arithmetic. We run an experimental campaign to confirm their observations and to measure the number of instructions devoted to WU and BN. In the case of ResNet-50, this instruction count is around 30 million instructions per batch, that is, just 0.04% of the total FP instructions. AlexNet and Inception V2 present similar results. In the case of the *seq2seq* model, this instruction count remains low, 1.60% of the total FP instructions. In conclusion, this data motivates our efforts to reduce the cost of FMA instructions



(a) Instruction mix (b) Static techniques on ResNet-50 (c) Static techniques on *seq2seq* model

Fig. 2: Analysis for evaluated DNNs

for DNN training, since WU calculations and BN layers, which must be computed using FP32 arithmetic, represent a negligible portion of the training workload.

B. Static Approaches for Training

SoA training methods [8] employ the same data representation format for a certain variable throughout the whole training process, hence the term static. Similarly, we also evaluate a static approach that performs all FMA instructions in full BF16 except for WU calculations and BN layers, which are performed in FP32. Figure 2b shows the three different static training techniques on ResNet-50. The methodology we use to generate Figure 2b is described in Section VI. We observe that MP achieves very similar accuracy as FP32. In contrast, the BF16 approach does not deliver the desired level of accuracy with a significant drop, around 3%, w.r.t FP32 and MP. Figure 2c shows the same three techniques on the *seq2seq* model. Further details on the model appear in Section VI. Again, FP32 and MP present similar validation loss curves, while the BF16 approach fails to deliver comparable results.

An FMA entirely relying on BF16 precision can potentially provide large performance improvements. However, as Figures 2b and 2c illustrate, full BF16 training fails to deliver competitive levels of accuracy.

C. FMAs Data Representation Requirements

We analyze FMA data representation requirements using ResNet-50 training in order to explain the accuracy drop of full BF16 in Figures 2b and 2c.

Since the addition step of FMA instructions requires right-shifting the smaller of the two operands by the difference of the exponents, it is possible to completely eliminate the smallest number if the exponent difference is larger than the amount of *mantissa* bits. In the case of BF16, there are 7 *mantissa* bits. This issue is called *swamping* [6].

Figure 3 shows the exponent differences for all FMA instructions involved in ResNet-50 training on different epochs. It represents the percentage of FMAs (*y-axis*) that avoid entirely losing one of the two operands in the addition for a determined number of *mantissa* bits (*x-axis*). During the first epoch, just around 60% of the FMA instructions would not entirely lose one of the two addition operands when using the BF16 data representation. For epochs 8, 16, and 32, this percentage is slightly lower. In contrast, 23 mantissa bits, i.e., the standard FP32 representation, allows to keep the two addition operands of near 100% of the FMAs.

This analysis clearly shows the reasons behind the lower accuracy observed when using BF16 FMA instructions. Consequently, blindly applying BF16 FMA across the training process of DNNs is not a viable training strategy, as it leads to widespread *swamping* issues.

IV. Dynamic Precision Training

We propose to dynamically switch floating point precision during training to obtain the same accuracy as FP32 or MP while reducing training costs by employing full BF16 FMAs for as many batches as possible. Our strategy for DNN training uses BF16 FMAs as long as the training loss improves. When we detect training loss stagnation, we switch to a higher precision approach such as MP or FP32 for a number of batches, until training loss improves again. The use of reduced precision helps with the generalization capabilities of the model [5]. To detect training loss stagnation we calculate its Exponential Moving Average (EMA) [17] for a moving window of batches. We compare this EMA value with a threshold parameter to determine if training is progressing with the currently employed FMA format. The EMA is updated throughout the training process, guiding the decisions to switch between full BF16 and MP precision.

Algorithm 1 shows the high level pseudo-code of our *Dynamic* precision training strategy. The algorithm starts the training process using the SoA MP training [8] for several batches, defined by $numBatchesMP$ parameter. Then, it computes the EMA of the training loss and, if its reduction is above a certain threshold ($emaThreshold$ parameter), it computes the next $numBatchesBF16$ using BF16 training. Once training has processed these $numBatchesBF16$ batches, our algorithm updates the EMA value and compares it against the $emaThreshold$ parameter again. If the EMA reduction is not large enough, the algorithm switches back to MP training. Otherwise, it keeps using BF16 training for $numBatchesBF16$ batches before updating the EMA value and comparing against it once again.

This method is applied during the entire training and it is able to dynamically adapt the way each batch handles its FMA instructions depending on the training loss progress. If such progress stagnates, the FMAs are computed using a more expensive higher-precision method like the MP or FP32

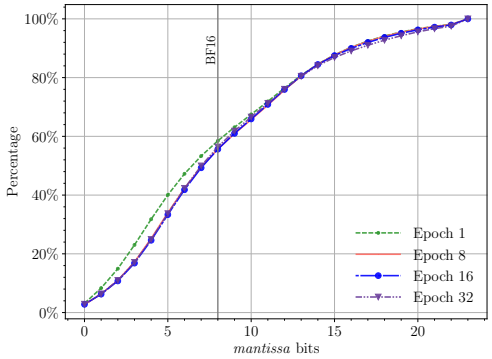


Fig. 3: Percentage of FMAs without *swamping*.

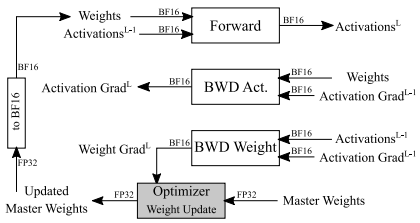


Fig. 4: BF16 training process for a layer.

techniques. If training progresses well, the algorithm dynamically switches the way FMAs are computed and chooses the full BF16 approach. Figure 4 illustrates the implications of processing one batch using BF16 FMAs. Similar to MP: (i) weights, activations, and gradients are stored in BF16 format; (ii) an FP32 master copy of the weights is maintained and updated with the weight gradients during the optimizer step (e.g., in the Stochastic Gradient Descent solver); and (iii) reduction operations present in BN layers use FP32 arithmetic. In contrast, all arithmetic operations within the forward and backward passes are performed using full BF16 arithmetic. Section VII demonstrates that dynamically switching the precision of batch execution between full BF16 and MP FMAs provides the same training convergence than FP32 or MP FMAs during the whole training achieving performance gains.

V. SERP: Seamless Emulation of Reduced Precision Formats

Due to the lack of available hardware implementing full BF16 FMAs, software emulation is required to study their numerical behavior. Several approaches have been used in the past to emulate BF16 arithmetic, most notably via highly-tuned low-level libraries that truncate floating point operands [8], [18]. However, such approaches require to access the mathematical library upon which the training relies. And this should be done for all potentially different implementation frameworks such as Tensorflow, PyTorch, or Caffe. With the aforementioned approaches these arithmetic modifications must be done on the source code or at compile time. Besides the effort that code modifications or recompilation of complex frameworks requires, this methodology is not applicable to closed-source mathematical libraries, like Intel MKL. Since computer arithmetic is highly sensible to the implementation,

Algorithm 1: Dynamic Precision Training

$numBatchesMP \leftarrow 10$ // Number of consecutive MP batches
$numBatchesBF16 \leftarrow 1000$ // Number of consecutive BF16 batches
$emaThreshold \leftarrow 0.04$ // Defines EMA reduction threshold

$precisionModeBF16 \leftarrow False$ // Indicates current precision mode, $True$ means *BF16*
$countBatchesBF16 \leftarrow 0$ // Counts how many *numBatchesBF16* have been executed
$numBatchesTrain \leftarrow numBatchesMP$ // Number of batches per training loop iteration

for $i = 0$ to $niter$ **do**
  train.step(numBatchesTrain) // *numBatchesTrain* batches *precisionModeBF16*
  $trainingLoss[i] \leftarrow train.trainingLoss$
  if $i = 5$ **then** // Initial history to calculate *EMA*
    $EMA \leftarrow average(trainingLoss)$
  if $i > 5$ **then**
    $EMAprev \leftarrow EMA$
    $EMA \leftarrow emaCalculation(trainingLoss, EMAprev)$ // Each *numBatchesMP*
    if $(precisionModeBF16! = True)$ **then**
      if $((EMAprev - EMA) > emaThreshold)$ **then** // If training loss goes down
        $precisionModeBF16 \leftarrow True$
        changeToBF16() // Switch precision to BF16
    else
      $countBatchesBF16 \leftarrow countBatchesBF16 + numBatchesTrain$
      if $(countBatchesBF16 = numBatchesBF16)$ **then**
        if $((EMAprev - EMA) > emaThreshold)$ **then** // If training loss goes down
          $countBatchesBF16 \leftarrow 0$ // Stay in BF16 precision
        else // If training loss stagnates
          $precisionModeBF16 \leftarrow False$
          changeToMP() // Switch precision to MP
          $countBatchesBF16 \leftarrow 0$

not instrumenting the true executed binary sequence of instruction can lead to wrong interpretation [19].

To overcome these limitations we developed SERP, a dynamic binary instrumentation tool based on Pin 3.7 [20]. SERP captures and instruments the dynamic instruction flow at runtime, which enables modifying the instruction operands to the targeted numerical data format without changing the application source code, effectively emulating lower precision numerical formats. Our approach seamlessly works on complex frameworks like PyTorch, Tensorflow, or Caffe, with interpreted languages, and is able to instrument instructions inside dynamically linked libraries.

SERP performs the following steps: First, it checks the current FMA operation mode, which for the purposes of this paper can be FP32, MP, or BF16 (see Figure 1). Additional numerical formats can be easily emulated. Second, it determines whether we are executing routines that belong to WU calculations or BN layers. If yes, computation proceeds in FP32. Third, the tool intercepts all floating point instructions of the workload, including FMAs. For each FMA instruction, SERP rounds off all operands that need to be converted to BF16 using an accurate round to nearest even algorithm. Finally, SERP can dynamically change its FMA operation mode via a simple inter-process communication method that can be invoked from the Python high-level DNN framework code were Algorithm 1 is implemented.

To mitigate the overhead of SERP, we implement two optimizations: First, we vectorize the truncation and rounding routines via AVX512 instructions. Second, we avoid redundant rounding and truncation operations by identifying instructions that: (i) belong to the same basic block, and (ii) share input operands already stored in the register file with the BF16 format. These two optimizations keep the overhead of SERP below 25× with respect to a native run.

VI. Experimental Methodology

A. Experimental Setup

Our experiments are performed on Intel Xeon Platinum 8160 processors, which implement the AVX512 ISA extension. To train and test the CNN implementations we use the Intel-Caffe [21] framework (version 1.1.6a). We use the Intel MKLDNN [22] (version 0.18.0) Deep Neural Network library, and the Intel MKL library(version 2019.0.3) to run numerical kernels since both libraries are optimized to run on our testing infrastructure. To define and run the experiments we use the *pyCaffe* python interface, which takes care of loading the data and orchestrating the execution. Finally, to perform the RNNs experiments we use PyTorch [23] (version 1.4.0), Intel MKLDNN (version 0.21.1), and the Intel MKL library (version 2019.4). We use torchtext to manage the preprocessing steps needed in the *seq2seq* model.

B. Neural Network Models

To evaluate our proposals we consider three representative SoA CNN models: AlexNet [1], Inception V2 [13], [14] and ResNet-50 [15]. These models are the backbone of recently published highly successful models [24].

We use the ImageNet database [25] as training input. To keep execution times manageable when using SERP, we run the experiments using a reduced ImageNet Database, similar to the Tiny ImageNet Visual Recognition challenge data set [26]. Therefore, we use 256,000 images divided into 200 categories for training, and 10,000 images for validation. The images have no modifications in terms of size. All the evaluated CNN models remain unmodified.

AlexNet is selected due to its simplicity in terms of structure and amount of required computations. To train AlexNet we consider a batch size of 256 and the base learning rate is 0.01, which is adjusted every 20 epochs taking into account a weight decay of 0.0005 and a momentum of 0.9. This model is trained for 32 epochs.

Inception V2 is a model conceived to reduce computational costs via cheap 1×1 convolutions. To train it we use a batch size of 64 and a base learning rate of 0.045, which is updated every 427 steps (0.11 epochs). The gamma, momentum and weight decay are set to 0.96, 0.9, and 0.0002, respectively. The training process is executed for 16 epochs.

ResNet-50 is a network that delivers good accuracy and avoids the vanishing gradients issue by using residual blocks. We train it using a multi-step approach. The batch size is 64 and the base learning rate is 0.05, which is updated every 30 epochs. The gamma hyperparameter, momentum value, and weight decay are set to 0.1, 0.9, and 0.0001, respectively. The training process runs for a total of 32 epochs.

Finally, to test the performance of our approach over RNNs we selected a *seq2seq* model to solve the NMT task [3]. This model is trained using the Multi30K dataset [27]. This dataset has 30,000 multilingual English-German sentences. We take 29,000 sentences as the training set and 1,000 for validation purposes. The model has 20,000,000 parameters in total, we use a batch size of 256, and the Adam Optimizer to do the training process. The model was built using Gated Recurrent Units (GRU). Training is done over 10 epochs.

C. Static and Dynamic Schemes

This paper considers two types of training techniques: **static schemes** and **dynamic schemes**. When using static schemes, training uses the same data representation form for a given parameter during its complete execution. For example, Figures 2b and 2c display results obtained using static schemes. We employ the following schemes:

- **MP:** This scheme replicates prior work on MP. FMA instructions that belong to WU calculations and BN layers always use FP32 precision. The remaining FMA instructions use the MP approach represented in Figure 1b.
- **BF16:** FMA instructions that belong to WU calculations and BN layers always use FP32 precision. The remaining FMA instructions use BF16 operands to multiply and to accumulate (Figure 1c).

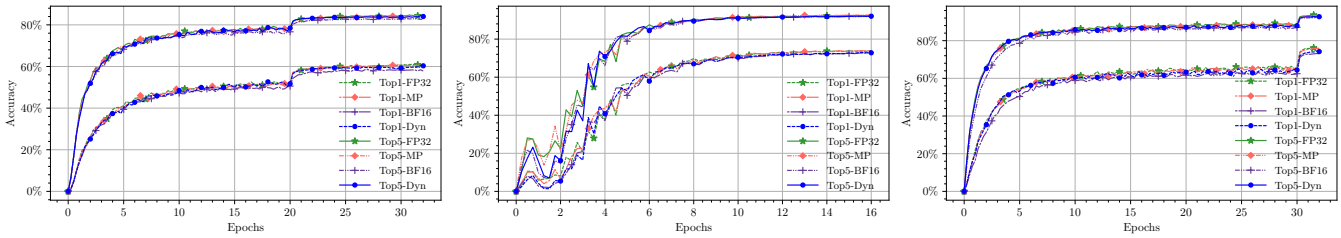
The *Dynamic* scheme we propose in this paper switches between the MP and BF16 static techniques during training, as explained in Section IV and detailed in Algorithm 1. This dynamic method aims to retain the favorable training convergence properties of MP, while relying on BF16 FMAs for a large portion of the execution.

To generate the results we show in Sections VII-A, VII-B, and VII-D we set parameters $emaThreshold$, $numBatchesBF16$, and $numBatchesMP$ to 0.04, 1000, and 10, respectively, when training AlexNet, Inception V2, and ResNet-50. The *seq2seq* model employs a different data set, Multi30K, which requires adapting $emaThreshold$, $numBatchesBF16$ and $numBatchesMP$ to the number of batches per epoch of the Multi30K training process. We set them to 0.06, 15 and 1, respectively. In addition, we run an experimental campaign considering different parameter configurations in Section VII-C.

D. Emulating BF16

There is no real hardware supporting full BF16 FMAs, which Figure 1c represents. To evaluate the numerical behavior of BF16 FMAs, we use our SERP tool, which is described in Section V. To evaluate performance improvement when using *Dynamic*, we attach the Sniper simulator [9] to SERP. Since both tools are based on Pin, combining them is natural.

Sniper is a high-speed and accurate x86 computer architecture simulator. We extend Sniper to support the AVX512 ISA, including its FMA instructions. We simulate a standard Xeon processor by considering the hardware parameters that Table II displays. We simulate with Sniper the execution of one training batch of ResNet-50 and *seq2seq* using FP32, MP and BF16 FMAs. Since SERP provides the percentage of batches that are computed in BF16 and MP during the whole training, we extrapolate the overall performance by using the performance metrics that Sniper provides. This approach is equivalent to the widely used SimPoints method [28] and makes it possible to estimate the performance of the whole training workload.



(a) AlexNet (b) Inception V2 (c) ResNet-50

Fig. 5: Test accuracy of evaluated training strategies.

TABLE II: Sniper Parameters

Component	Description
CPU	2.1 GHz, Out-of-Order
ITLB	128-entries, 4-associativity
DTLB	64-entries, 4-associativity
STLB	512-entries 4-associativity
L1 ICache	32 KB, 4-associativity, 1-shared cores
L1 DCache	32 KB, 8-associativity, 1-shared cores
L2 Cache	1 MB, 8-associativity, 1-shared cores
L3 Cache	32 MB, 16-associativity, 24-shared cores
Bandwidth	30 GB/s per core

TABLE III: Accuracy and percentage of FMAs executed in *BF16* precision.

Model	Epoch	FP32		MP		Dynamic			BF16		
		Top-1	Top-5	Top-1	Top-5	Top-1	Top-5	BF16FMA	Top-1	Top-5	BF16FMA
AlexNet	32	60.79%	84.50%	60.18%	84.43%	60.32%	84.02%	94.60%	57.80%	82.56%	99.93%
Inception	16	74.01%	92.36%	73.73%	92.67%	72.80%	92.02%	95.55%	72.03%	92.05%	99.90%
ResNet-50	32	75.96%	93.37%	75.70%	93.20%	74.20%	92.70%	96.40%	72.97%	92.30%	99.91%

VII. Evaluation

A. Convolutional Neural Networks

Figure 5 shows the validation accuracy of the three considered CNN models for the different training strategies. The *x-axis* represents the epochs of the training process while the *y-axis* shows the accuracy reached by the model over the validation set. In addition, Table III shows the *Top-1* and *Top-5* validation accuracies reached on the three network models for each training strategy, along with the percentage of FMA instructions fully computed with BF16 operands. We consider the FP32, MP, and BF16 static strategies and our *Dynamic*.

Figure 5a shows that BF16 displays worse accuracy than *Dynamic* or MP. Table III shows that FP32, MP, *Dynamic*, and BF16 reach *Top-5* accuracies of 84.50%, 84.43%, 84.02% and 82.56% respectively after 32 epochs. Importantly, *Dynamic* reaches comparable accuracy with respect to FP32 and MP while performing 94.60% of the FMA instructions in full BF16 precision. In contrast, BF16 training does 99.93% of the FMAs in full BF16 precision (0.07% are in WU and BN layers), but the accuracy drops by almost 3% in *Top-1* and 2% in *Top-5*. This significant drop in accuracy takes place by performing an additional 5% of BF16 FMAs compared to *Dynamic*. This shows that our proposal successfully achieves SoA accuracy levels while still relying mostly on BF16 FMAs.

Figure 5b shows validation accuracy over 16 epochs for the Inception V2 model. Accuracy fluctuates initially due to its structure and recommended hyperparameters. *Dynamic* responds in a robust way to these changes, which highlights its general applicability. Table III shows that FP32, MP, *Dynamic*, and BF16 reach *Top-5* accuracies of 92.36%, 92.67%, 92.02%, and 92.05% respectively after 16 epochs. BF16 training is able to achieve SoA accuracy since this network is designed to be robust to noise and tolerates lower precision.

The evaluation on ResNet-50 (Figure 5c) demonstrates that *Dynamic* training is effective when applied to deeper CNNs. In this case, the accuracy of the model reaches SoA levels while using BF16 for 96.40% of the FMA instructions. Table III displays the accuracy numbers we obtain from our evaluation after 32 epochs. BF16 training fails to deliver SoA *Top-1* accuracy with a drop of 2.99% with respect to FP32. *Dynamic* is able to close the accuracy gap between FP32 and BF16 training by performing a large percentage of FMA instructions in BF16 precision.

In summary, *Dynamic* is able to achieve comparable accuracy with respect to FP32 and MP training while performing $\geq 94.60\%$ of the FMA instructions in BF16 precision.

B. Sequence to Sequence RNN Model

Figure 6 shows the *seq2seq* validation loss achieved over 10 epochs for all the considered strategies. Table IV contains the final numbers for training and validation loss, and the percentage of BF16 precision FMAs.

While FP32, MP, and *Dynamic* perform similarly for *seq2seq*, BF16 is not able to yield comparable validation loss. Table IV shows that *Dynamic* achieves SoA accuracy in terms of both validation loss (Val-Loss) and training loss (Tr-Loss) while performing 66.0% of the FMA instructions fully using BF16. In addition, the loss function values for validation data slightly improve in MP and *Dynamic* with respect to FP32. This behavior has been studied [29].

C. Sensitivity Analysis for Dynamic Precision Algorithm

We perform a sensitivity analysis of the parameters employed in Algorithm 1. We consider *numBatchesBF16* = {500, 1000, 2000}, and *emaThreshold* = {0.02, 0.04, 0.08}, while *numBatchesMP* is set to 10. Figure 9 shows, for a number of ResNet-50 epochs, the accuracy obtained for each of the 9 tested configurations. In addition, we include the accuracy values of BF16, MP, and FP32. The accuracy of all *Dynamic* configurations is above BF16. The most relevant parameter is *emaThreshold*, as it decides when to switch between different FMA approaches. As long as this parameter is reasonably set

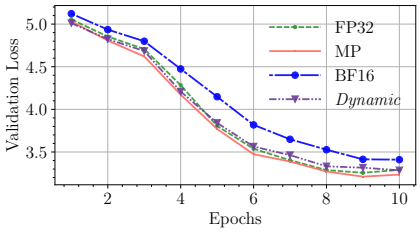


Fig. 6: Validation Loss of the *seq2seq*.

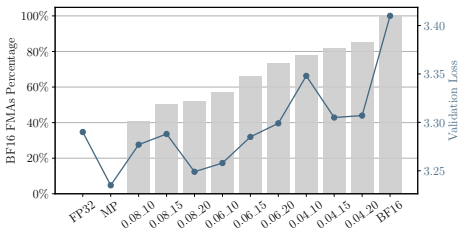


Fig. 7: Sensitivity analysis of *seq2seq*.

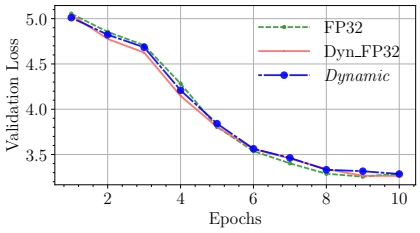


Fig. 8: Val. Loss of *Dyn_FP32*

TABLE IV: Loss and FMAs executed in BF16 precision for *seq2seq*

Model	Epoch	FP32		MP		Dynamic			BF16		
		Tr-Loss	Val-Loss	Tr-Loss	Val-loss	Tr-Loss	Val-Loss	BF16FMA	Tr-Loss	Val-Loss	BF16FMA
seq2seq	10	2.019	3.290	2.008	3.235	2.122	3.285	66.0%	2.392	3.410	99.5%

TABLE V: Performance results

Model	Batch execution time (s)			Training Speed-up w.r.t FP32	
	FP32	MP	BF16	FP32-BF16	MP-BF16
ResNet-50	23.80	22.15	17.11	1.38×	1.39×
seq2seq	45.73	41.17	34.40	1.22×	1.26×

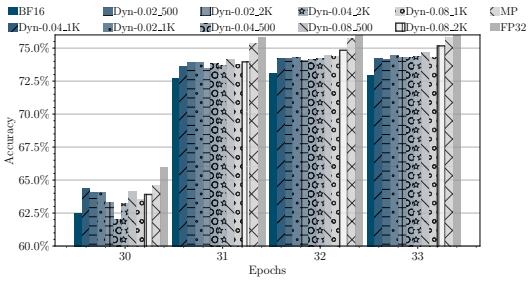


Fig. 9: Sensitivity analysis of *Dynamic* on ResNet-50.

to detect training loss improvement or degradation, *Dynamic* achieves SoA accuracy.

Figure 7 shows the sensitivity analysis for the *seq2seq* model. We consider *numBatchesBF16* = {10, 15, 20}, and *emaThreshold* = {0.04, 0.06, 0.08}, while *numBatchesMP* is set to 1. Bars show the percentage of FMAs run using BF16 format per each parameter configuration, and the line represents its corresponding validation loss. We find that validation loss improves the FP32 value for some configurations. This is due to the numerical noise injected by reduced data representation formats, which may help the training processes in some scenarios. Previous work has also observed this effect [29]. Multiple *Dynamic* training configurations obtain SoA validation loss with up to 66% BF16 FMAs.

D. Dynamically Switching Between FP32 and BF16

The evaluation of Sections VII-A, VII-B, and VII-C considers a *Dynamic* approach that switches between MP and BF16 FMAs. However, the *Dynamic* technique can also switch between FP32 and BF16. Figure 8 shows validation loss evolution while training *seq2seq* considering a *Dyn_FP32* approach, which switches between FP32 and BF16. Results obtained with *Dyn_FP32* do not present any noticeable deviation to the ones obtained with *Dynamic* in terms of accuracy and percentage of BF16 FMA instructions. A similar behavior is observed with the CNN models. This certifies that our proposal can be applied in machines that support FP32 and BF16 FMAs, without needing MP.

E. Sniper Results

We evaluate the performance of BF16 training by attaching the SERP tool to the Sniper simulator. Table V shows the training time of one batch using FP32, MP, and full BF16 FMAs. In addition, we show the speed-up of *Dynamic* (MP-BF16) and *Dyn_FP32* (FP32-BF16) w.r.t FP32 training. We use ResNet-50 and *seq2seq* as example models.

As can be seen in the table, the execution time of one batch using BF16 is significantly faster than FP32 or MP. This is due to lower memory bandwidth requirements and doubling the FMA vectorization (AVX512) throughput, as having all input and output operands in 16 bits enables more data level parallelism. The speed-ups achieved for the entire training process with the *Dynamic* approach reach 1.39× and 1.26× for ResNet-50 and *seq2seq*, respectively.

VIII. RELATED WORK

Reducing training requirements via lower precision data types has been an active topic in recent years. Numerous proposals use non-standard data formats with ad-hoc bit widths for exponent and mantissa, which lack hardware support but enable going as low as 8-bits for some computations. The use of stochastic rounding (SR) techniques also proves effective but can be costly to implement in hardware and software.

For example, prior work indicates that dynamic fixed-point is sufficient to train DNNs with low precision multipliers [4]. This approach obtains SoA results by uniformly applying the dynamic fixed point format with different scaling factors driven by the overflow rate displayed by the fixed-point numbers. Applying SR to 16-bit fixed-point multiply and add operators has been proven to be beneficial [5]. Other proposals train DNNs using 8-bit floating point (FP8) numbers by relying on a combination of 8-bit and 16-bit arithmetic and using SR to obtain SoA results [6], or propose a multi-precision approach also using SR to keep DNN training accuracy [30]. Finally, there is a newer approach that does not require SR [31]; however, a quantization step and two newly defined FP8 numerical data types are needed.

Other proposals focus on leveraging available hardware support to achieve the same objectives. This is the case of

the MP proposals that employ half-precision formats such as FP16 and BF16 for tensors [7], [8]. The BF16 numerical format has been used in specific-purpose hardware targeting DNNs [32], and will soon be supported by off-the-shelf hardware from Intel and Arm. Our dynamic training approach falls under this category. As we demonstrate, it can be applied on top of FP32 or MP training to reduce memory bandwidth and computational requirements, while delivering comparable accuracy. Furthermore, SERP and the *Dynamic* approach we proposed can be adapted and used to explore and validate other existing and future encodings.

IX. CONCLUSIONS

This paper analyzes the instruction mix of DNN training workloads. We show that FMA instructions represent the 60% of these workloads. While MP training can deliver SoA accuracy, training schemes that rely on half-precision FMAs, like BF16 training, fail to deliver comparable accuracy levels. We propose a *Dynamic* training technique that can perform a large portion of FMAs in full half-precision, lowering training requirements without hurting accuracy. We achieve this by training in BF16 mode and identifying when training convergence stagnates, at which point we switch to a higher precision strategy like MP or FP32 until stagnation dissipates.

We evaluate our proposal considering three SoA CNNs and one RNN model. We use SERP to instrument all FMA instructions and modify operands to the targeted numerical data type. We demonstrate that half-precision BF16 can be used extensively on $\geq 94.6\%$ of all FMAs during the training of deep CNN models, and on 66.0% of FMAs in our evaluated *seq2seq* model, while reaching comparable accuracy levels with respect to FP32 and MP training. Finally, our performance evaluation shows that the *Dynamic* approach can achieve speed-ups of up to $1.39\times$ with respect to FP32.

ACKNOWLEDGMENTS

Marc Casas has been partially supported by the Spanish Ministry of Economy, Industry and Competitiveness under Ramon y Cajal fellowship number RYC-2017-23269. Adrià Armejach has been partially supported by the Spanish Ministry of Economy, Industry and Competitiveness under Juan de la Cierva postdoctoral fellowship number IJCI-2017-33945. John Osorio has been partially supported by the Spanish Government by FPI pre-doctoral scholarship number PRE2019-090406 under project SEV-2015-0493-19-4. This work has been partially supported by Intel under the BSC-Intel collaboration and PE085800 - DEEP-SEA EU project.

REFERENCES

[1] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *NeurIPS*, 2012.

[2] Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, and K. Macherey, "Google's neural machine translation system: Bridging the gap between human and machine translation," *arXiv preprint*, 2016.

[3] D. Bahdanau, K. H. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," *ICLR*, 2015.

[4] M. Courbariaux, Y. Bengio, and J. David, "Training deep neural networks with low precision multiplications," *ICLR*, 2015.

[5] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, "Deep learning with limited numerical precision," *CoRR*, 2015.

[6] N. Wang, J. Choi, D. Brand, C. Y. Chen, and K. Gopalakrishnan, "Training DNN with 8-bit floating point numbers," *NeurIPS*, 2018.

[7] P. Micikevicius, S. Narang, J. Alben, G. Diamos, E. Elsen, D. Garcia, B. Ginsburg, M. Houston, O. Kuchaiev, G. Venkatesh, and H. Wu, "Mixed Precision Training," *ICLR*, 2018.

[8] D. Kalamkar, D. Mudigere, N. Mellempudi, D. Das, K. Banerjee, S. Avancha, D. T. Vooturi, N. Jammalamadaka, J. Huang, H. Yuen, J. Yang, J. Park, A. Heinecke, E. Georganas, S. Srinivasan, A. Kundu, M. Smelyanskiy, B. Kaul, and P. Dubey, "A Study of BFLOAT16 for Deep Learning Training," 2019.

[9] T. E. Carlson, W. Heirman, and L. Eeckhout, "Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulations," in *SC*, Nov. 2011, pp. 52:1–52:12.

[10] Nvidia. (2020, may) Nvidia a100 tensor core gpu architecture. [Online]. Available: https://bit.ly/2YAXLLY

[11] N. P. Jouppi, C. Young, N. Patil, D. Patterson, and et. al., "In-datacenter performance analysis of a tensor processing unit," in *44th ISCA*, 2017.

[12] Intel. (2020) Intel architecture instruction set extensions and future features programming reference. [Online]. Available: https://intel.ly/3czWOpS

[13] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and R. A., "Going deeper with convolutions," in *CVPR*, 2015.

[14] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," 2015.

[15] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," *CoRR*, 2015.

[16] B. Trevett. (2020, jan) Neural machine translation by jointly learning to align and translate. [Online]. Available: https://bit.ly/38lD6v4

[17] A. J. Lawrance and P. A. W. Lewis, "An exponential moving-average sequence and point process," *Journal of Applied Probability*, 1977.

[18] Y. Chatelain, E. Petit, P. de Oliveira Castro, G. Lartigue, and D. Defour, "Automatic exploration of reduced floating-point representations in iterative methods," in *Euro-Par*, 2019.

[19] C. Denis, P. de Oliveira Castro, and E. Petit, "Verificarlo: Checking floating point accuracy through monte carlo arithmetic," in *ARITH 2016*.

[20] C. K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," *ACM SIGPLAN Notices*, 2005.

[21] Intel. Intel caffe framework optimization. [Online]. Available: https://github.com/intel/caffe

[22] ——. Intel deep neural network library. [Online]. Available: https://github.com/intel/mkl-dnn

[23] A. Paszke, S. Gross, S. Chintala, and G. Chanan. (2020) Pytorch. [Online]. Available: https://github.com/pytorch/pytorch

[24] T. J. Yang, M. D. Collins, Y. Zhu, J. J. Hwang, T. Liu, X. Zhang, V. Sze, G. Papandreou, and C. L.C., "Deeperlab: Single-shot image parser," *CoRR*, 2019.

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

[26] L. Fei-Fei. Tiny imagenet visual recognition challenge. [Online]. Available: https://tiny-imagenet.herokuapp.com/

[27] D. Elliott, S. Frank, K. Sima'an, and L. Specia, "Multi30k: Multilingual english-german image descriptions," in *Proceedings of the 5th Workshop on Vision and Language*, 2016.

[28] G. Hamerly, E. Perelman, J. Lau, and B. Calder, "Simpoint 3.0: Faster and more flexible program analysis," in *Journal of Instruction Level Parallelism*, 2005.

[29] K. Audhkhasi, O. Osoba, and B. Kosko, "Noise benefits in backpropagation and deep bidirectional pre-training," in *IJCNN*, 2013.

[30] A. Rajagopal, D. A. Vink, S. I. Venieris, and C.-S. Bouganis, "Multi-precision policy enforced training (muppet): A precision-switching strategy for quantised fixed-point training of cnns," 2020.

[31] X. Sun, J. Choi, C. Y. Chen, N. Wang, S. Venkataramani, V. Srinivasan, X. Cui, W. Zhang, and K. Gopalakrishnan, "Hybrid 8-bit Floating Point (HFP8) Training and Inference for Deep Neural Networks," in *NeurIPS*, 2019.

[32] S. Wang and P. Kanwar, "Bfloat16: The secret to high performance on cloud tpus," 2019. [Online]. Available: https://bit.ly/2SlSYqq