A BF16 FMA is All You Need for DNN Training

John Osorio, Adrià Armejach, Eric Petit, Greg Henry, and Marc Casas

Abstract—Fused Multiply-Add (FMA) functional units constitute a fundamental hardware component to train Deep Neural Networks (DNNs). Its silicon area grows quadratically with the mantissa bit count of the computer number format, which has motivated the adoption of the BrainFloat16 format (BF16). BF16 features 1 sign, 8 exponent and 7 explicit mantissa bits. Some approaches to train DNNs achieve significant performance benefits by using the BF16 format. However, these approaches must combine BF16 with the standard IEEE 754 Floating-Point 32-bit (FP32) format to achieve state-of-the-art training accuracy, which limits the impact of adopting BF16. This paper proposes the first approach able to train complex DNNs entirely using the BF16 format. We propose a new class of FMA operators, $\text{FMA}^{\text{bf16}}_{\text{n\_m}}$, that entirely rely on BF16 FMA hardware instructions and deliver the same accuracy as FP32. $\text{FMA}^{\text{bf16}}_{\text{n\_m}}$ operators achieve performance improvements within the 1.28-1.35× range on ResNet101 with respect to FP32. $\text{FMA}^{\text{bf16}}_{\text{n\_m}}$ enables training complex DNNs on simple low-end hardware devices without requiring expensive FP32 FMA functional units.

Index Terms—Neural Nets, machine learning, reduced precision, FMA Operators, BF16, FP32, swamping, computer arithmetic, emulation, hardware.

1 Introduction

Fused Multiply-Add (FMA) functional units compute the operation $D = A \cdot B + C$ and constitute a key hardware component to train Deep Neural Networks (DNNs), since they support the vast majority of floating-point instructions required for DNN training [1]. FMA input datatypes and, in particular, their *mantissa* bit counts drive the performance and the hardware cost of FMA units. Indeed, the silicon area of FMA units feature a quadratic growth with respect to the number of *mantissa* bits. For example, a 16-bit multiplier using 8 *mantissa* bits requires $8^2 = 64$ units of area, while a 32-bit multiplier considering 24 *mantissa* bits employs $24^2 = 576$ units [2], [3], [4]. This advantage in terms of area budget has motivated the adoption of the BrainFloat16 format (BF16), which features 1 sign, 8 exponent, and 7 explicit mantissa bits, by many hardware vendors [2], [5]. Specialized hardware units targeting FMA computations based on the BF16 format deliver more floating point throughput than FP32 units while using the same silicon area [2], [5].

Despite these advantages, the potential of using BF16 during the entire training process is not fully exploited by any proposal. Previous work has described the numerical issues of BF16, which are caused by its reduced mantissa bits budget [6], [7]. To overcome them, previous approaches combine the BF16 format with the standard IEEE 754 32-bit Floating-Point (FP32) format when computing FMA instructions. The accumulation step uses FP32 arithmetic to combine $A \cdot B$ with C, and represent the final output D. In addition, some frequently used routines like Weight Update (WU) operations or Batch Normalization (BN) must entirely rely on the FP32 format to achieve state-of-the-art accuracy [6], [7].

- *J. Osorio, A. Armejach and M. Casas are with the Department of Computer Science, Barcelona Supercomputing Center, Barcelona, CA, 08034. And with Computer Architecture program, Universitat Politècnica de Catalunya, Barcelona, CA, 08034. E-mail: john.osorio@bsc.es*
- *E. Petit and G. Henry are with Intel Corporation, Portland, OR 97124.*

Manuscript received April 19, 2005; revised August 26, 2015.

There are more aggressive proposals that use 4-bit (FP4) datatypes for ultra-low precision training [8] or that dynamically employ between 3 and 8 bits of precision [9], [10]. However, these proposals require tailored hardware, extensive modifications to software frameworks, and very complex ad-hoc steps to guarantee training convergence. In addition, these techniques require 32-bit precision floating-point arithmetic for WU or FMA accumulators. In addition, these proposals do not achieve FP32 training accuracy.

This paper proposes the first approach to train state-of-the-art DNNs entirely using the BF16 format, without code and hyper-parameters tuning, while delivering the same accuracy as FP32. We propose a new class of FMA operators, $\text{FMA}^{\text{bf16}}_{\text{n\_m}}$, that entirely rely on BF16-based datatypes for its inputs and outputs. When computing an FMA operation, $\text{FMA}^{\text{bf16}}_{\text{n\_m}}$ represents input operands A and B using N BF16 literals, which we call BF16xN representations, while input C and output D use M BF16 literals, i.e. BF16xM types. $\text{FMA}^{\text{bf16}}_{\text{n\_m}}$ operators can be used for the entire training process of DNNs, effectively removing the need for FP32 architectural support at the hardware and ISA levels.

To evaluate the numerical behavior of $\text{FMA}^{\text{bf16}}_{\text{n\_m}}$ we implement Serp (Seamless Emulation of Reduced Precision Formats), a binary analysis tool. Serp uses Intel Pin [11] to intercept FMA instructions and modify its floating-point operands. Serp works on major software frameworks like Tensorflow, Caffe, or PyTorch seamlessly. Using Serp we evaluate 7 $\text{FMA}^{\text{bf16}}_{\text{n\_m}}$ variants on a wide range of DNNs: ResNet (18, 34, 50, 101) [12] and MobileNetV2 [13] on the CIFAR10 and CIFAR100 datasets, LSTMx2 model on Penn Treebank (PTB) dataset [14], two transformer-based models using IWSLT16 [15] and Multi30K dataset [16], [17] , and the Neural Graph Collaborative Filtering (NGCF) [18] a recommender system applied to the MovieLens: ML-100k dataset [19] We demonstrate that $\text{FMA}^{\text{bf16}}_{\text{n\_m}}$ achieves the same accuracy as FP32 training, while never resorting to FP32 FMA computations.

In addition, we perform micro-architectural simulations

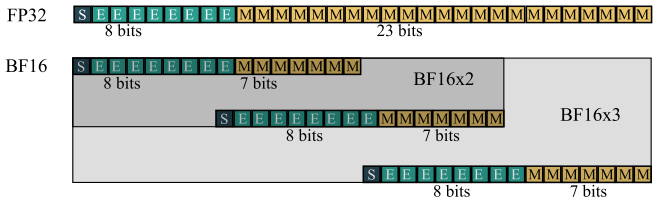


Fig. 1: BF16xN Data Formats.

using Snipersim [20] to evaluate the performance of $FMA^{bf16}_{n\_m}$ operators when replacing 32-bit FMA functional units on a Skylake-like processor. Our evaluation shows that $FMA^{bf16}_{1\_2}$ and $FMA^{bf16}_{2\_2}$ reach 1.34× and 1.28× speed-up on ResNet101, respectively, with respect to FP32 at equivalent area.

To the best of our knowledge, $FMA^{bf16}_{n\_m}$ is the first proposal enabling the exclusive use of BF16 arithmetic while achieving the same accuracy properties as FP32 for a large variety of networks, while providing performance improvements within the 1.28-1.34× range with respect to FP32. Thereby, enabling training of complex DNNs on simple low-end hardware devices by lifting the requirement of having FP32 hardware support and expensive FP32 FMA units.

2 The BF16xN Data Representation

The BF16xN data representation format is a compound datatype composed of N BF16 literals. It is a generalization of BF16x3, which was proposed by Henry et al. [2] to replace FP32 in High-Performance Computing (HPC) workloads. The BF16x1 format uses 1-bit and 8-bits storage for sign and exponent, like FP32, and 7 explicit mantissa bits. As shown in Figure 1, by concatenating three BF16 numbers we have 24 bits to represent the mantissa, 7 explicit bits and 1 implicit bit per BF16 literal, which is equivalent to the FP32 mantissa (23 explicit and 1 implicit bits). In this paper we consider BF16x3 and two more computer number formats based on the BF16xN compound datatypes: BF16x1 and BF16x2.

To describe the conversion from FP32 to BF16xN, we define the conversion operand $BF(\cdot)$ as the rounding process of an FP32 expression to BF16 via the Rounding to Nearest Even (RNE) algorithm. The $BF(\cdot)$ operand converts a generic FP32 value to its BF16x2 representation via the first two equations of Formula 1, or to its BF16x3 compound datatype representation using the three equations of Formula 1 [2]. The BF16 expression a_0 contains the same sign and exponent bits as the FP32 number a plus its top 7 mantissa bits. The 8th mantissa bit of a_0 is defined by RNE. Similarly, a_1 contains in its mantissa the second set of 8 bits of the BF16xN expression, which is at least 8 bits away of a_0 least significant bit. Finally, a_2 contains in its mantissa the third set of 8 bits which are at least 8 and 16 bits away of a_0 and a_1 least significant bit, respectively. BF16xN expressions are trivially converted back to FP32 by accumulating the a_i values on a FP32 register.

$$\begin{aligned} a_0 &= BF(a) \\ a_1 &= BF(a - a_0) \\ a_2 &= BF(a - (a_0 - a_1)) \end{aligned} \tag{1}$$

These equations are still valid for FP32 number $a = 0$, in that case all a_i terms will be zeros. However we need a special case for $+/-Inf$. In this case, to avoid NaN we will set all a_i to the corresponding infinity. We cannot use $a_1 = a_2 = 0$ since it will lead to NaN values in later Mul/FMA operations while the FP32 value would have produced the expected $+/-Inf$.

2.1 Computing FMA instructions with BF16xN

An FMA with BF16xN datatypes for the $a \cdot b$ product can be reduced to a set of partial products using BF16x1. Equation 2 shows the FMA operation $c := c + a \cdot b$ where a and b are represented as BF16x3, i.e., $a := a_0 + a_1 + a_2$ and $b := b_0 + b_1 + b_2$; and c in FP32. The FMA is expressed as nine partial products and the addition of c [2].

$$\begin{aligned} c := c + a \cdot b = c &+ a_0 \cdot b_0 + a_0 \cdot b_1 + a_0 \cdot b_2 \\ &+ a_1 \cdot b_0 + a_1 \cdot b_1 + a_2 \cdot b_0 \\ &+ a_1 \cdot b_2 + a_2 \cdot b_1 + a_2 \cdot b_2 \end{aligned} \tag{2}$$

Henry et al. [2] analyze the error of the BF16x3 format and demonstrate that accumulating the six most significant partial products is enough to keep FP32 precision for a wide range of values. Indeed, the $a_1 \cdot b_2$ and $a_2 \cdot b_1$ products most significant bits are at least 24 bits behind the result's most significant bits. For the same reasons, the $a_2 \cdot b_2$ product is at least 32 bits behind. Their sum is therefore 23 bits behind the result's most significant bits. Therefore, the maximum level of error of a BF16x3 FMA compared to a FP32 FMA is in the same order of magnitude as round-off effects.

Based on these observations, the three least significant multiplications (see Equation 3) can be avoided, while providing accuracy within the $[23, 24]$ bit range.

$$\begin{aligned} c := c + a \cdot b \approx c &+ a_0 \cdot b_0 + a_0 \cdot b_1 + a_0 \cdot b_2 \\ &+ a_1 \cdot b_0 + a_1 \cdot b_1 + a_2 \cdot b_0 \end{aligned} \tag{3}$$

BF16x2 decomposes an FP32 number into two BF16 literals, which provides an intermediate format between BF16x1 and FP32 with significantly lower compute cost than FP32 and similar bandwidth requirements. When multiplying two BF16x2 numbers, we have to compute four partial multiplications, as Equation 4 shows.

$$\begin{aligned} a \cdot b = (a_0 + a_1) \cdot (b_0 + b_1) \\ a \cdot b = a_0 \cdot b_0 + a_0 \cdot b_1 \\ a_1 \cdot b_0 + a_1 \cdot b_1 \end{aligned} \tag{4}$$

Following a similar reasoning as Henry et al. [2] with BF16x3, we can drop the last term, $a_1 \cdot b_1$ while still keeping $[15, 16]$ bits of mantissa accuracy for a wide range of values.

Henry et al. [2] require FP32 support for the accumulator input c, and for the conversion between FP32 and BF16xN. Our proposed $FMA^{bf16}_{n\_m}$ operators, introduced in Section 4, do not require FP32 support as all input and output operands are expressed as BF16 literals.

3 Suitability of BF16xN for DNN Training

This section illustrates how BF16xN delivers the required accuracy to enable DNN training using BF16 arithmetic exclusively, including critical routines like Weight Updates (WU) or Batch Normalization (BN).

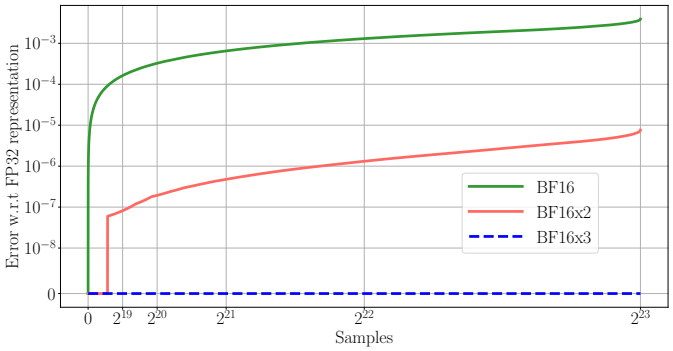


Fig. 2: Error with respect to FP32 representation for different BF16xN formats for all possible *mantissa* combinations over a fixed exponent (2^{23} samples).

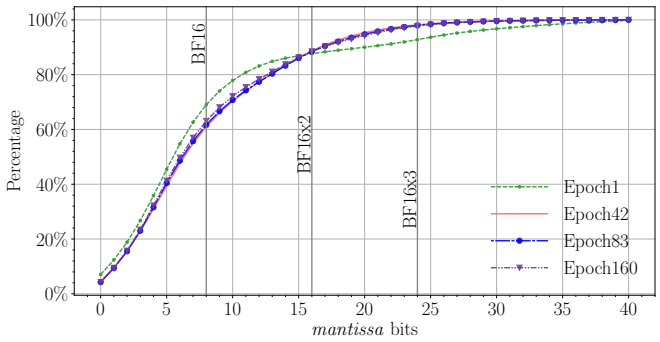


Fig. 3: Percentage of FMAs without *swamping* for a given number of *mantissa* bits.

3.1 BF16xN Representation Errors

Figure 2 shows numerical errors of BF16x1, BF16x2, and BF16x3 FMAs, when representing all the 2^{23} possible representations of FP32 explicit mantissa bits. Without lose of generality, we consider the exponent value to be fixed at 2. The *x-axis* represents the 2^{23} samples sorted in terms of the absolute numerical error of BF16x1, BF16x2, and BF16x3 when representing them. The *y-axis* displays the magnitude of such error.

When using BF16x1, only 3.84% of the samples experience an error below 10^{-4}. This representation error has been shown in prior works to prevent DNN training convergence when applied to the entire training process [6], [7]. When employing BF16x2, 41.95% of the samples display errors below 10^{-6}. The remaining 58.05% present errors between 10^{-6} and 10^{-5}. Figure 2 shows how the BF16x2 error is consistently 3 orders of magnitude lower than BF16x1. As Section 7 demonstrates, this low error enables full DNN training with the BF16x2 datatype. Finally, the BF16x3 format has no significant error compared to FP32. Indeed, it has enough *mantissa* bits to exactly represent all FP32 samples.

3.2 *Swamping* Issues in DNN Training

During the accumulation phase of FMA instructions, the *mantissa* of the smallest value shifts according to the exponent difference between the operands. This shift brings the possibility to eliminate the smallest operand when the exponent difference is bigger than the number of *mantissa* bits. This full absorption issue is known as *swamping* [4] in the deep learning community.

Figure 3 shows the percentage of FMAs without *swamping* issues on the *y-axis* for a given number of accumulator *mantissa* bits, represented in the *x-axis*. We collect this data when training ResNet101 on CIFAR100 across different epochs. Section 6 describes in detail our experimental setup. We show that *swamping* would appear on 40% of the FMA operations when using BF16x1 on the accumulator input, on 12% with BF16x2, and nearly 0% with BF16x3. Our experiments evaluate the impact of *swamping* and reveal a correlation between *swamping* prevalence and the applicability of BF16xN data types.

Diminishing the number of *mantissa* bits makes DNNs more sensitive to *swamping* and may lead to critical information loss. In fact, the use of BF16x1 leads to a substantial amount of *swamping* that prevents reaching state-of-the-art training accuracy levels [4]. The BF16x2 and BF16x3 data formats entirely rely on BF16 arithmetic functional units and do not suffer from the common numerical issues of BF16x1 that Figures 2 and 3 display.

4 FMA Operators Based on BF16 Arithmetic

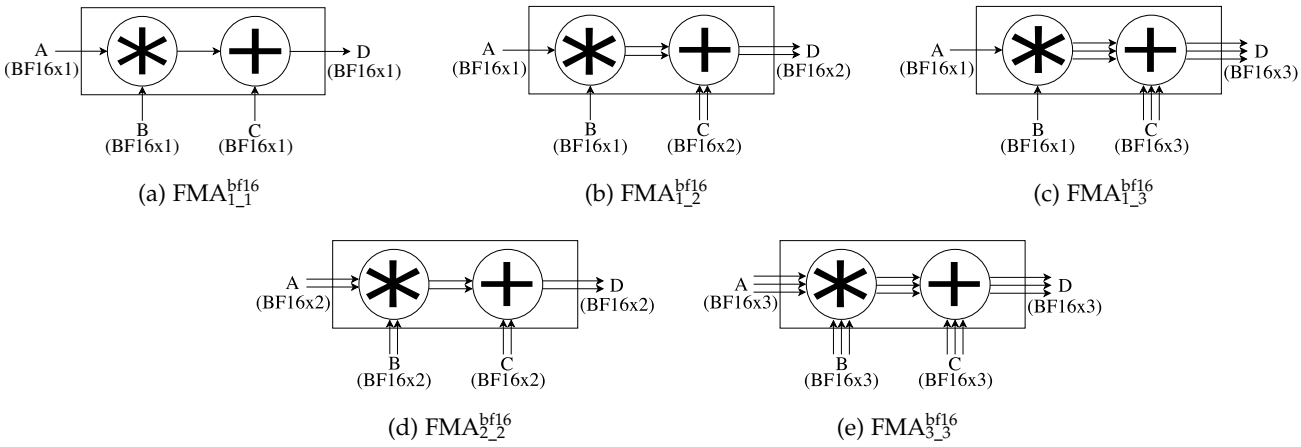
This section introduces a new class of FMA operators, $\text{FMA}^{\text{bf16}}_{\text{n\_m}}$, that entirely rely on BF16-based operands. When performing an FMA operation $D = A \cdot B + C$, $\text{FMA}^{\text{bf16}}_{\text{n\_m}}$ represents operands A and B via the BF16xN format, while it uses a representation with a potentially different number of BF16 literals, BF16xM, for C and D.

4.1 The $\text{FMA}^{\text{bf16}}_{\text{n\_m}}$ operators

We propose and evaluate four new $\text{FMA}^{\text{bf16}}_{\text{n\_m}}$ operators: $\text{FMA}^{\text{bf16}}_{\text{1\_2}}$, $\text{FMA}^{\text{bf16}}_{\text{1\_3}}$, $\text{FMA}^{\text{bf16}}_{\text{2\_2}}$, and $\text{FMA}^{\text{bf16}}_{\text{3\_3}}$. The first two operators, which are represented in Figures 4b and 4c, use the BF16x1 representation for inputs A and B. The $\text{FMA}^{\text{bf16}}_{\text{1\_2}}$ operator uses the BF16x2 number format for parameters C and D, while $\text{FMA}^{\text{bf16}}_{\text{1\_3}}$ uses BF16x3 to represent them. The other two $\text{FMA}^{\text{bf16}}_{\text{n\_m}}$ operators use compound data types in all of their inputs. We term these two operators $\text{FMA}^{\text{bf16}}_{\text{2\_2}}$ (Figure 4d) and $\text{FMA}^{\text{bf16}}_{\text{3\_3}}$ (Figure 4e). As Section 2 describes, the $\text{FMA}^{\text{bf16}}_{\text{2\_2}}$ operator can be implemented using three or four partial products. We denote these two variants as $\text{FMA}^{\text{bf16}}_{\text{2\_2}}$ $\{3\}$ and $\text{FMA}^{\text{bf16}}_{\text{2\_2}}$ $\{4\}$, respectively. Equation 5 defines $\text{FMA}^{\text{bf16}}_{\text{2\_2}}$ with four products. In addition, $\text{FMA}^{\text{bf16}}_{\text{3\_3}}$ can consider six or nine partial products, as Equations 3 and 2 indicate. We call these operators $\text{FMA}^{\text{bf16}}_{\text{3\_3}}$ $\{6\}$ and $\text{FMA}^{\text{bf16}}_{\text{3\_3}}$ $\{9\}$.

$$c = c + a \cdot b \approx (c_0 + c_1) + a_0 \cdot b_0 + a_0 \cdot b_1 \\ a_1 \cdot b_0 + a_1 \cdot b_1 \quad (5)$$

Figure 4 defines the semantics of the operators in terms of input and output datatypes. The internal implementation of the operators can be done in different ways, and it is orthogonal to the underlying hardware and ISA definitions. For example, a possible implementation for the $\text{FMA}^{\text{bf16}}_{\text{2\_2}}$ operator is to perform the four partial products, as shown in



(a) $FMA^{bf16}_{1\_1}$ (b) $FMA^{bf16}_{1\_2}$ (c) $FMA^{bf16}_{1\_3}$

(d) $FMA^{bf16}_{2\_2}$ (e) $FMA^{bf16}_{3\_3}$

Fig. 4: $FMA^{bf16}_{n\_m}$ Operators

TABLE 1: $FMA^{bf16}_{n\_m}$ characterization in terms of mantissa bits, bitwidth, number of BF16 multiplications, area units required to implement the entire operator, and expected speed-up over FP32 FMAs at equivalent area.

$FMA^{bf16}_{n\_m}$	$FMA^{bf16}_{1\_1}$	$FMA^{bf16}_{1\_2}$	$FMA^{bf16}_{1\_3}$	$FMA^{bf16}_{2\_2}$ {3}	$FMA^{bf16}_{2\_2}$ {4}	$FMA^{bf16}_{3\_3}$ {6}	$FMA^{bf16}_{3\_3}$ {9}	FP32
Multiplier *mantissa* bits	8	8	8	$[15, 16^{*}]$	16	$[23, 24^{**}]$	24	24
Maximum input bitwidth	16	32	48	32	32	48	48	32
# BF16 multiplications	1	1	1	3	4	6	9	N/A
# Area Units	64	64	64	192	256	384	576	576
Speed-up wrt FP32 (equivalent area)	9.0×	9.0×	9.0×	3.0×	2.3×	1.5×	1.0×	1.0×

* For a number $> 2^{-111}$; ** For a number $> 2^{-103}$

Equation 4, and then perform an intermediate accumulation step to reduce the number of BF16 literals from four to two, which matches the data type of the other input of the accumulator (C), as depicted in Figure 4. These internal accumulation step does not require FP32 arithmetic as it exclusively involves BF16 literals. Finally, the last accumulation step can be done by adding BF16 literals of each input in pairs, where each literal addition produces an output literal of D.

Our experimental campaign in Section 7 demonstrates that $FMA^{bf16}_{n\_m}$ operators achieve comparable accuracy with respect to FP32 executions on large and complex DNNs. Our $FMA^{bf16}_{n\_m}$ operators entirely use BF16 arithmetic, that is, they do not employ FP32 computations on any layer, including BN, Softmax, and WU routines.

4.2 Characterization of $FMA^{bf16}_{n\_m}$ units

To characterize our $FMA^{bf16}_{n\_m}$ units we use the observation that the area of an FMA is dominated by the multiplier as it grows quadratically with mantissa bits [3], [4]. An FP32 FMA requires $24^2 = 576$ area units, while an FMA with BF16 multiplier inputs would require just $8^2 = 64$ units. Therefore, BF16 FMAs are 9.0× smaller than FP32 FMAs. Moreover, the use of dense hardware units (e.g., NVIDIA's tensor cores [21] or Google's TPU [22]) leads to efficient matrix compute engines that can deliver 8-32× more FLOP/S than equivalent FP32 hardware [2]. For example, NVIDIA A100's can deliver up to 19.5 TFLOP/S in FP32, but when using BF16 tensor cores they reach 312 TFLOP/S peak throughput, i.e., 16.0× more FLOP/S.

Table 1 characterizes $FMA^{bf16}_{n\_m}$ operators in terms of: the number of multiplier *mantissa* bits, the maximum bitwidth of input parameters, the number of BF16 partial multiplications required by the corresponding $FMA^{bf16}_{n\_m}$ unit, the number of area units required to implement the operator, and the attainable theoretical speed-up in compute throughput at equivalent area with respect to FP32. In the case of $FMA^{bf16}_{1\_1}$, $FMA^{bf16}_{1\_2}$ and $FMA^{bf16}_{1\_3}$ the peak floating-point throughput gain with respect to FP32 hardware is 9.0× since we can accommodate 9 FMA BF16 functional units in the area of a single FP32 FMA unit. In the case of $FMA^{bf16}_{2\_2}$ {3} and $FMA^{bf16}_{2\_2}$ {4} three and four BF16 products are required, respectively. Therefore, their maximum theoretical throughput is 3.0× and 2.3× larger than FP32 FMAs, respectively. When considering $FMA^{bf16}_{3\_3}$ {6} and $FMA^{bf16}_{3\_3}$ {9}, six and nine BF16 products are required, which means these units deliver 1.5× and 1.0× more floating-point throughput than a single FP32 unit, respectively, with the same area.

While Table 1 characterizes $FMA^{bf16}_{n\_m}$ units in terms of their maximum floating-point throughput, the performance they reach when training DNNs depends on many other factors. For example, operators employing BF16x3 will require more memory traffic and register storage than those using BF16x2. Section 7 provides performance in terms of total elapsed time with respect to FP32 using an accurate micro-architectural simulator.

5 SERP: AN $FMA^{BF16}_{N\_M}$ EMULATION TOOL

To assess the numerical properties of DNN training workloads when they are exposed to the $FMA^{bf16}_{n\_m}$ units, we develop SERP (Seamless Emulation of Reduced Precision Formats), a binary analysis tool based on Intel Pin 3.7 [11]. There is no hardware supporting FMA units relying entirely

TABLE 2: Instruction mix on three well-known DNN workloads.

Model	Other FP	FP FMA	Non FP
AlexNet	1.02%	57.42%	41.54%
Inception	1.13%	60.93%	37.94%
ResNet	1.19%	62.95%	35.86%

on the BF16 format. Indeed, hardware products that support the BF16 format use mixed-precision FMA units that combine BF16 with higher precision inputs (e.g., FP32) [22], [23], [24]. Alternatively, [25] and [6] use highly-tuned low-level implementations applied at the source code level to modify floating-point instructions. However, these approaches require extensive modifications of the complex mathematical libraries supporting DNN frameworks like Tensorflow, PyTorch, or Caffe. Besides these difficulties, the modification of proprietary mathematical libraries like Intel MKL [26] is not possible for most users as the code is not publicly available.

5.1 SERP Design

Our goal is to use SERP to perform fast, accurate and seamless emulation of $\text{FMA}^{\text{bf16}}_{\text{n\_m}}$ operators. In addition, we want to be able to emulate code of external dynamically linked libraries, as many applications rely on such libraries which contain key optimized routines. Therefore, we propose to leave the target application unmodified and operate at binary level intercepting the executed machine instructions. By identifying key floating point instructions, for which we can modify the input and output operands, SERP can seamlessly work on any application and DNN framework including dynamically linked external libraries.

Table 2 shows the instruction mix breakdown for three well-known DNN workloads. As can be seen, the bulk of floating-point instructions are of the type FMA. Therefore, continuing with the previous example, if we are able to identify and instrument these instructions from the dynamic execution instruction flow we can perform fine-grain emulation of $\text{FMA}^{\text{bf16}}_{\text{n\_m}}$ operations seamlessly.

5.2 SERP Implementation

SERP relies on dynamic binary translation (DBT) to modify the dynamic instruction flow of the application binary. These modifications are done during the *instrumentation* step, which is executed only once.

SERP is configured through a simple configuration file that specifies the desired parameters in terms of routines and instructions to be instrumented as well as the emulated reduced precision format and rounding method. The step that performs code instrumentation goes through each statically defined basic block once, and for each instruction of interest SERP inserts the code. In our context, we want to perform the following checks:

1) **Before the instruction is executed:** Insert code that converts the source registers of the instruction to the desired BF16xN depending on the $\text{FMA}^{\text{bf16}}_{\text{n\_m}}$ operator emulated.

Listing 1: Code to convert an FP32 FMA to $\text{FMA}^{\text{bf16}}_{\text{2\_2}}$ {3}

```
1  inline void process_operands(__m512* operand1,
2              __m512* operand2, __m512* operand3,
3              __m512* destination)
4  {
5      __m512 sumind2, sumind3, sum1, sum2;
6      __m512 multiplier, sumfull;
7
8      // Multiplier inputs
9      __m512 sx1_op1, sx2_op1;
10     __m512 sx1_op2, sx2_op2;
11
12     // Accumulator input
13     __m512 sx1_op3, sx2_op3;
14
15     // Multiplier result
16     __m512 sx1_mult, sx2_mult;
17
18     // Convert multiplier inputs to BF16x2 with RNE
19     convert_float_to_two_bfloats_vec(operand1,
20             (__m512*)&sx1_op1, (__m512*)&sx2_op1);
21     convert_float_to_two_bfloats_vec(operand2,
22             (__m512*)&sx1_op2, (__m512*)&sx2_op2);
23
24     //Convert the accumulator input to BF16x2 with RNE
25     convert_float_to_two_bfloats_vec(operand3,
26             (__m512*)&sx1_op3, (__m512*)&sx2_op3);
27
28     //Multiplication with 3 partial products
29     sumind3 = _mm512_mul_ps(sx1_op1, sx2_op2);
30     sumind3 = _mm512_fmadd_ps(sx2_op1, sx1_op2, sumind3);
31     sumind2 = _mm512_mul_ps(sx1_op1, sx1_op2);
32     multiplier = _mm512_add_ps(sumind3, sumind2);
33
34     //Convert multiplier output to BF16x2 with RNE
35     convert_float_to_two_bfloats_vec((__m512*)&multiplier,
36                 (__m512*)&sx1_mult, (__m512*)&sx2_mult);
37
38     // Addition
39     sum1 = _mm512_add_ps(sx1_mult, sx1_op3);
40     sum2 = _mm512_add_ps(sx2_mult, sx2_op3);
41     sumfull = _mm512_add_ps(sum1, sum2);
42     *destination = sumfull;
43 }
```

2) **Instruction:** In some cases, the instruction can be executed as is with the modified source registers, for example for $\text{FMA}^{\text{bf16}}_{\text{1\_1}}$. In other cases, when the numerical format will not execute as expected on the existing instruction or available hardware, the instruction needs to be replaced by equivalent code that emulates the intended behaviour. For example, when emulating the $\text{FMA}^{\text{bf16}}_{\text{2\_2}}$ operator the semantic of the instruction changes.
3) **After the instruction is executed:** Insert code that converts the output to the desired output format.

Listing 1 shows the C++ source code injected during the instrumentation of AVX512 FMA instructions when emulating the $\text{FMA}^{\text{bf16}}_{\text{2\_2}}$ {3} operator. Lines 5-16 define temporary variables used for the multiplication and accumulation steps. SERP uses Equation 1 in lines 18-26 to convert each input operand (a, b, and c) to its BF16x2 representation. Lines 29-31 implement the 3 partial products using the BF16 literals, which means we drop the last term: $a_1 \cdot b_1$ (see Equation 5). The variable *multiplier* in line 32 is the output of the multiplier within the FMA operator. In line 35 we convert the *multiplier* output back to $\text{FMA}^{\text{bf16}}_{\text{2\_2}}$ to proceed with the accumulation step, that adds the BF16 literals from the multiplier output with input c in lines 39 and 40. Finally, line 41 packs the two resulting BF16 literals into an FP32 representation in order to store the result back into the *destination* FP32 register of the original instrumented instruction (line 42).

Once the code has been instrumented at basic block

level, the next step is *analysis*. During the analysis step the instrumented dynamic instruction flow, which includes external libraries, is executed. Analysis is the most compute expensive step as the modified instruction flow with code insertions is executed.

SERP seamlessly works on frameworks such as Tensorflow, PyTorch or Caffe without requiring any source code modification. SERP is open source and can be found with concrete installation and execution steps at:
https:\\urlredacted.

6 EXPERIMENTAL METHODOLOGY

Our experimental methodology considers the $FMA^{bf16}_{1\_1}$, $FMA^{bf16}_{1\_2}$, $FMA^{bf16}_{1\_3}$, $FMA^{bf16}_{2\_2}$ $\{3\}$, $FMA^{bf16}_{2\_2}$ $\{4\}$, $FMA^{bf16}_{3\_3}$ $\{6\}$, and $FMA^{bf16}_{3\_3}$ $\{9\}$ operators. In contrast with prior proposals [6], [7], we do not employ FP32 precision on any routines to improve training convergence. Therefore, when using $FMA^{bf16}_{n\_m}$ operators, all FMA instructions use BF16 arithmetic for the whole training process. We compare the training accuracy of $FMA^{bf16}_{n\_m}$ approaches against two baselines: A full FP32 training and an approach that uses mixed-precision FMAs, described in Section 6.6. The latter uses FP32 accumulators during the whole training process, and full FP32 FMAs to process batch normalization (BN) layers and compute weight updates (WU).

6.1 Object Classification Models

We consider the following object classification models: ResNet18, ResNet34, ResNet50, ResNet101 [12], and MobilenetV2 [13] on both CIFAR10 and CIFAR100 datasets [27]. We use the hyperparameter setup recommended by the state-of-the-art [8]. We train all networks for 160 epochs using the SGD Optimizer. We use a batch size equal to 128, an initial learning rate equal to 0.1, which we divide by a factor of 10 at epochs 82 and 122. We use momentum equal to 0.9 and a weight decay of 10^{-4}. For ResNet18, we do not use the weight decay value.

6.2 Natural Language Processing Models

We consider three natural language processing models:

The LSTMx2 model [14] applied to the PTB dataset. We train it for a total of 39 epochs and use a batch size of 20, an initial learning rate equal to 1, two LSTM layers, a hidden size of 650, a sequence length of 35, and a dropout equal to 0.5. We follow the details in [14] to train the medium size model; our test uses the source code available in [28].

A transformer-based model [15] applied to the IWSLT16 dataset to translate between Dutch and English. We train the model termed *base* for 20 epochs using the Adam Optimizer with $\beta_1 = 0.9$, $\beta_2 = 0.98$, and $\epsilon = 10^{-9}$. We use a batch size of 12000 and 4000 warm-up steps. We use the source code available in [29]. All additional details are in [15], [29].

Finally, we train a simple transformer-based model on the Multi30k dataset [16] to translate between English and Dutch. We train this model for ten epochs using the Adam optimizer with a fixed learning rate equal to $5 \cdot 10^{-4}$ and a batch size equal to 128. This implementation uses the source code available in [30].

6.3 Recommender System

We consider a recommender system based on Graph Neural Networks (GNN), the Neural Graph Collaborative Filtering (NGCF) [18]. This model is applied to the MovieLens: ML-100k dataset [19]. We train it for a total of 400 epochs with a batch size of 1024 and a learning rate of 0.0001. The number of embeddings is 64. Additional details regarding this model are described in the literature [31].

6.4 Training Accuracy of $FMA^{bf16}_{n\_m}$ Operators

Our training accuracy experiments using SERP run on a node with two 24-core Intel Xeon Platinum 8160 processors with AVX512 support. To train and validate ResNet, LSTMx2, the transformer-based models and the recommender system we use a source code compiled version of PyTorch [32] (version 1.8.0), Intel MKLDNN [33] (version 1.22.0) and the Intel MKL library [26] (version 2019.4).

6.5 Performance of $FMA^{bf16}_{n\_m}$

We evaluate the impact of using $FMA^{bf16}_{n\_m}$ operators in terms of performance using the Snipersim micro-architectural simulator [20]. While SERP enables highly accurate analysis on the ability of $FMA^{bf16}_{n\_m}$ operators to successfully train state-of-the-art DNN networks, it does not provide any information in terms of performance. Sniper implements a realistic processor model from which we can estimate the performance benefits of using $FMA^{bf16}_{n\_m}$. Both Sniper and SERP use Intel Pin 3.7 [11], which enables an easy interaction between them.

We extend Sniper to support the AVX512 ISA, including its FMA instructions. We simulate a standard Xeon processor by considering the hardware setup that Table 3 details. We consider the execution of one training batch of ResNet101 on CIFAR10; a Transformer based model on the Multi30K dataset and a LSTMx2 model on the PTB dataset. We use the FP32 baseline and the $FMA^{bf16}_{1\_1}$, $FMA^{bf16}_{1\_2}$, and $FMA^{bf16}_{2\_2}$ $\{4\}$ operators.

Since all $FMA^{bf16}_{n\_m}$ operators rely on BF16 arithmetic, our AVX512 model assumes BF16 support per each 16-bit lane. To simulate the additional throughput achievable for each $FMA^{bf16}_{n\_m}$ operator at equivalent FP32 area, we model wider functional units using the throughput numbers from Table 1. Architectural implications, e.g. memory bandwidth requirements, of having such wider functional units are taken into account. We coalesce up to 32 16-bit FMA instructions into a single 512-bit FMA instruction, which we send to the out-of-order core pipeline. These coalesced instructions fetch the required amount of data from memory, and go through the pipeline fulfilling all the dependencies of the original individual FMA instructions.

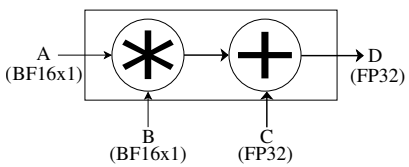


Fig. 5: Mixed Precision *Fused Multiply-Add* (FMA).

TABLE 3: Simulation parameters

Component	Simulated parameter
Core	2.1 GHz out-of-order, 192 entries reorder buffer, 2 x AVX512 FP32 FMAs
L1 ICache	32KB, 4-way associative, private
L1 DCache	32KB, 8-way associative, private
L2 Cache	1MB, 8-way associative, private
L3 Cache	32MB, 16-way associative, shared (24 cores)
Bandwidth	30 GB/s per core

6.6 Mixed-Precision FMA

Our evaluation considers the Mixed-Precision (MP) FMA instruction used by the Intel Advanced Matrix Extensions (Intel AMX) [34] or the Nvidia Ampere architecture [5]. Figure 5 illustrates an MP FMA. It considers the BF16 format for inputs A and B, and FP32 for input C and output D. Having a 32-bit output to store the accumulation of $A \cdot B$ and C reduces the risk of numerical hazards related to *swamping* [4]. MP training also employs full FP32 FMAs to process BN and WU layers.

7 Evaluation

We evaluate training accuracy of the $FMA^{bf16}_{1\_1}$, $FMA^{bf16}_{1\_2}$, $FMA^{bf16}_{1\_3}$, $FMA^{bf16}_{2\_2}$ {3}, $FMA^{bf16}_{2\_2}$ {4}, $FMA^{bf16}_{3\_3}$ {6}, and $FMA^{bf16}_{3\_3}$ {9} operators when training object classification and natural language processing models in Sections 7.1 and 7.2, respectively. We also consider FP32 and MP, which we describe in Section 6. Section 7.4 evaluates the performance improvements of $FMA^{bf16}_{n\_m}$ operators with respect to FP32.

7.1 Training Accuracy: Object Classification

Figure 6 shows the MobileNetV2 validation accuracy on CIFAR100 when using FP32, MP and seven $FMA^{bf16}_{n\_m}$ operators. The *x-axis* represents the epoch count while the *y-axis* shows the top1 accuracy achieved by the model over the validation set. FP32, MP, $FMA^{bf16}_{1\_2}$ and $FMA^{bf16}_{1\_1}$ obtain accuracies of 75.04%, 75.16%, 74.85% and 73.92%, respectively. The $FMA^{bf16}_{1\_1}$ approach fails to deliver similar accuracy as FP32. In contrast, $FMA^{bf16}_{1\_2}$ outperforms $FMA^{bf16}_{1\_1}$ and reaches similar accuracy as FP32 and MP. Higher precision operators like $FMA^{bf16}_{2\_2}$ {3} or $FMA^{bf16}_{3\_3}$ {6} reach 74.82% and 75.31% accuracy, respectively. They obtain similar or better accuracy than $FMA^{bf16}_{1\_2}$.

Figure 7 shows our evaluation results considering the ResNet18 model using the CIFAR100 dataset. $FMA^{bf16}_{1\_1}$ displays similar accuracy as FP32, it is just a 0.45% lower in validation accuracy. The other approaches also match FP32 accuracy. These results indicate that not very deep networks do not require the most accurate $FMA^{bf16}_{n\_m}$ versions to be trained. Figure 8 shows results for ResNet34. In this case, the $FMA^{bf16}_{1\_1}$ approach loses almost 1.0% compared to FP32, while $FMA^{bf16}_{1\_2}$ outperforms the FP32 approach by 0.73%. As networks become deeper, a half-precision (BF16) accumulator fails to deliver state-of-the-art accuracy, while $FMA^{bf16}_{n\_m}$ operators using at least BF16x2 to store accumulations obtain the same accuracy levels as FP32. This can be clearly observed when considering the validation accuracy that $FMA^{bf16}_{n\_m}$ approaches achieve when training the ResNet50 and ResNet101 models, which are displayed in Figures 9 and 10, respectively. $FMA^{bf16}_{1\_1}$ accuracy is significantly lower than FP32.

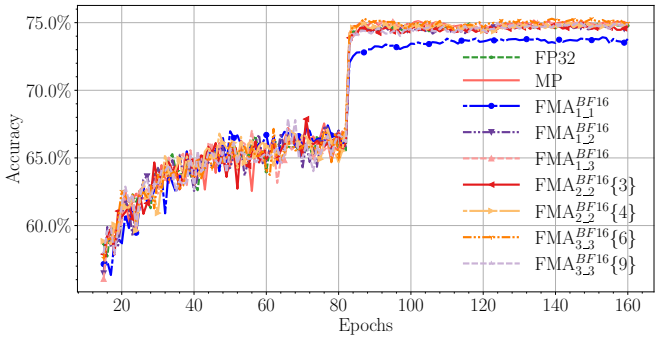


Fig. 6: MobilenetV2 Accuracy on CIFAR100 Validation Set.

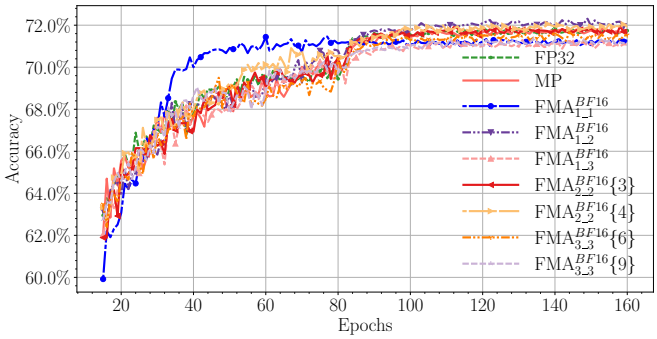


Fig. 7: ResNet18 Accuracy on CIFAR100 Validation Set.

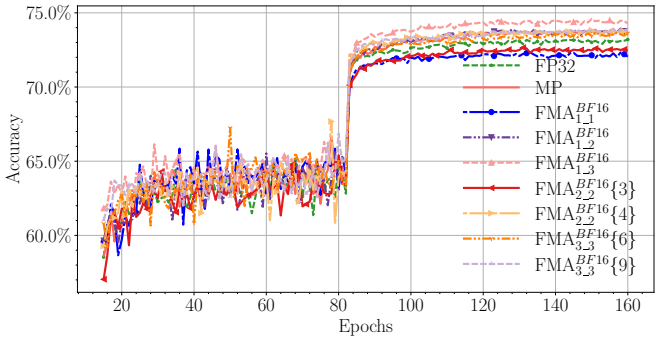


Fig. 8: ResNet34 Accuracy on CIFAR100 Validation Set.

In ResNet50 $FMA^{bf16}_{1\_2}$ already achieves similar accuracy when compared to FP32 and MP. In contrast, in ResNet101 $FMA^{bf16}_{1\_2}$ does have a slight drop with respect to FP32, from 75.93% to 73.75%, respectively. However, the use of $FMA^{bf16}_{2\_2}$ {3} leads to a training accuracy of 76.00%, which is again on par with that obtained with FP32.

Additionally, using CIFAR10 we obtain similar results. We again observe that as networks become deeper, more precision is needed to obtain the same levels of accuracy as FP32 or MP. Figure 11 shows the results training ResNet34 using the CIFAR10 dataset, again the trend is that operators using more accumulator bits obtain better accuracy results. In this specific case, $FMA^{bf16}_{1\_2}$ attains a validation accuracy

This article has been accepted for publication in IEEE Transactions on Emerging Topics in Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TETC.2022.3187770

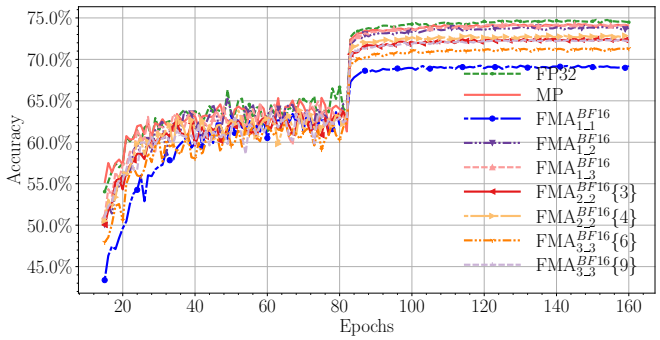


Fig. 9: ResNet50 Accuracy on CIFAR100 Validation Set.

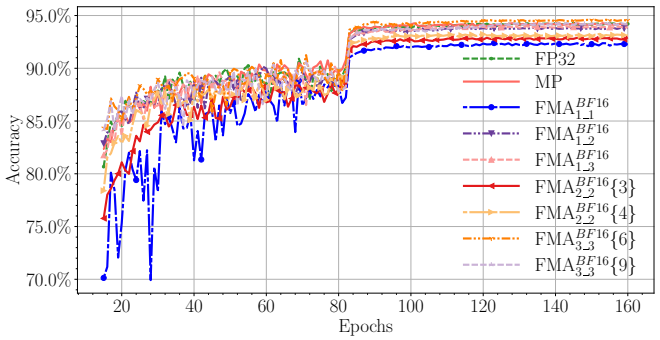


Fig. 11: ResNet34 Accuracy on CIFAR10 Validation Set.

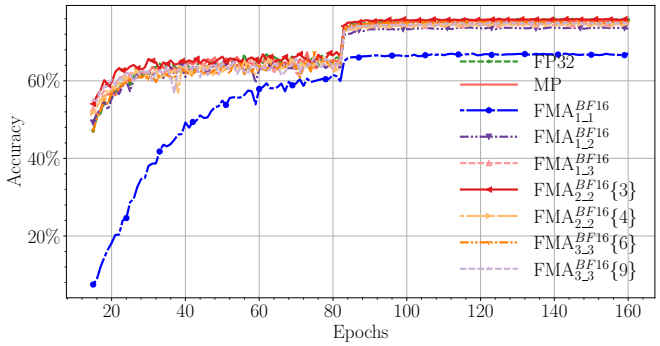


Fig. 10: ResNet101 Accuracy on CIFAR100 Validation Set.

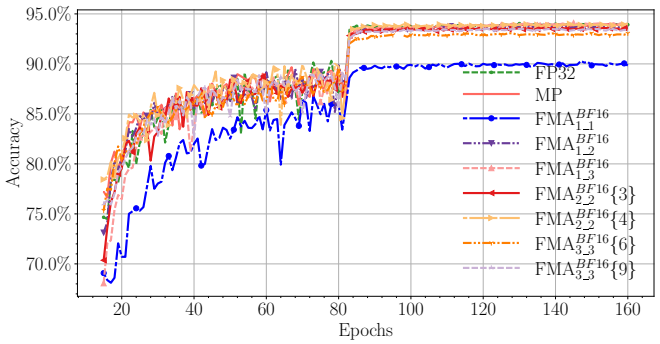


Fig. 12: ResNet50 Accuracy on CIFAR10 Validation Set.

of 93.86%, which is comparable to that achieved by FP32 (94.30%). In ResNet50 with CIFAR10, Figure 12, FMA$^{bf16}_{1_3}$ obtains the best accuracy results, 94.10%, being on par with respect to FP32. However, this operator needs to use 48 bits to save the result, which consumes additional storage and bandwidth. The FMA$^{bf16}_{2_2}$ operator displays the same levels of accuracy (94.05%) with better trade-offs in terms of storage and bandwidth requirements. Finally, similar results are obtained in ResNet101 (Figure 13), where the FMA$^{bf16}_{1_2}$ operator gives an accuracy result of 94.31% while FP32 obtains a 94.71%.

Tables 4 and 5 summarize the maximum top1 validation accuracy achieved by FP32, MP, and the 7 FMA$^{bf16}_{n_m}$ approaches on the 4 ResNet models and MobileNetV2 for the CIFAR10 and CIFAR100 datasets, respectively. We observe that FMA$^{bf16}_{1_1}$ fails to deliver the same accuracy as FP32 for the deepest models, i.e., ResNet34, ResNet50, and ResNet101. FMA$^{bf16}_{1_2}$ also achieves worse accuracy than FP32, particularly for the case of ResNet101 and the CIFAR100 data set. In contrast, the two FMA$^{bf16}_{2_2}$ variants behave very similarly as FP32 and MP. The very deep nature of ResNet101 requires an operator like FMA$^{bf16}_{2_2}$ {3}, which increases the representation accuracy of some FMA input parameters.

Some high-precision FMA$^{bf16}_{n_m}$ operators sometimes behave worse than others even if they have larger numerical precision. Audhkhasi et al. [35] explain this effect where noise can speed-up convergence during backpropagation. This can be observed in ResNet50 with CIFAR100 where FMA$^{bf16}_{2_2}$ {4} has lower accuracy than FP32 and FMA$^{bf16}_{1_2}$; however, in ResNet101 FMA$^{bf16}_{2_2}$ {3} achieves the highest accuracy.

In conclusion, FMA$^{bf16}_{n_m}$ operators, which are entirely based on BF16 FMA units, have the capacity to deliver comparable training accuracy with respect to FP32. In particular, the 2 FMA$^{bf16}_{2_2}$ variants deliver comparable state-of-the-art accuracy with respect to FP32 while potentially providing better performance. Section 7.4 evaluates FMA$^{bf16}_{2_2}$ {4} in terms of performance.

7.2 Training Accuracy: Natural Language Processing

We consider three natural language processing models in this section. Figure 14 shows the LSTMx2 model training process on the PTB data set. The *y-axis* represents test perplexity and the *x-axis* displays epoch count. The FMA$^{bf16}_{1_1}$ operator fails to converge, giving a NaN output after epoch thirteen. However, the other techniques obtain similar test perplexities as FP32, which is equal to 83.17. We run an additional experiment to explain the low accuracy of the FMA$^{bf16}_{1_1}$ operator. We compute the portion of FMA operations that do not suffer from numerical *swamping* effects at different epochs during the training of the LSTMx2 model. Figure 15 shows in the *y-axis* the percentage of FMA operations not suffering from *swamping* and in the *x-axis* the number of *mantissa* bits of the FMA accumulator input operand. We display results considering 4 different epochs. With the FMA$^{bf16}_{1_1}$ format (8 *mantissa* bits) only around 55% of the FMAs have no *swamping* for epochs 10, 12, and 13. This leads to a catastrophic loss of information after epoch 13, as we see in the test perplexity metric of Figure 14. However, with an operator like FMA$^{bf16}_{1_2}$ (16 *mantissa* bits in the accumulator), 85% of FMAs present no *swamping* at epoch 12, which allows training to complete successfully as shown before.

The second NLP model we consider is a transformer-based DNN trained to solve a Neural Machine Translation

TABLE 4: Accuracy on Validation Set for CIFAR10

Model	FP32	MP	$FMA_{1\_1}^{bf16}$	$FMA_{1\_2}^{bf16}$	$FMA_{1\_3}^{bf16}$	$FMA_{2\_2}^{bf16}$ {3}	$FMA_{2\_2}^{bf16}$ {4}	$FMA_{3\_3}^{bf16}$ {6}	$FMA_{3\_3}^{bf16}$ {9}
ResNet18	92.62%	92.99%	92.22%	92.31%	92.01%	92.86%	92.55%	92.16%	92.58%
ResNet34	94.30%	94.28%	92.39%	93.86%	94.00%	92.92%	93.24%	94.60%	94.29%
ResNet50	94.03%	94.00%	90.28%	93.77%	94.10%	93.66%	94.05%	93.07%	93.51%
ResNet101	94.71%	94.73%	91.19%	94.31%	94.30%	93.68%	94.07%	93.49%	93.31%
MobileNetV2	93.58%	93.91%	93.11%	93.93%	94.09%	93.70%	94.06%	93.95%	93.94%

TABLE 5: Accuracy on Validation Set for CIFAR100

Model	FP32	MP	$FMA_{1\_1}^{bf16}$	$FMA_{1\_2}^{bf16}$	$FMA_{1\_3}^{bf16}$	$FMA_{2\_2}^{bf16}$ {3}	$FMA_{2\_2}^{bf16}$ {4}	$FMA_{3\_3}^{bf16}$ {6}	$FMA_{3\_3}^{bf16}$ {9}
ResNet18	71.91%	71.89%	71.46%	72.31%	71.30%	71.95%	72.06%	71.67%	71.36%
ResNet34	73.21%	73.86%	72.83%	73.94%	74.59%	72.66%	73.87%	73.68%	73.94%
ResNet50	74.78%	74.25%	69.24%	73.93%	74.24%	72.57%	72.91%	71.32%	72.35%
ResNet101	75.93%	75.65%	67.10%	73.76%	74.65%	76.00%	74.75%	75.19%	74.98%
MobileNetV2	75.04%	75.16%	73.92%	74.85%	75.08%	74.82%	75.04%	75.31%	74.99%

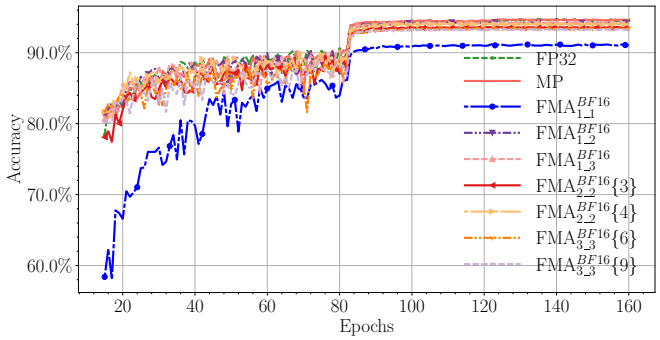


Fig. 13: ResNet101 Accuracy on CIFAR10 Validation Set.

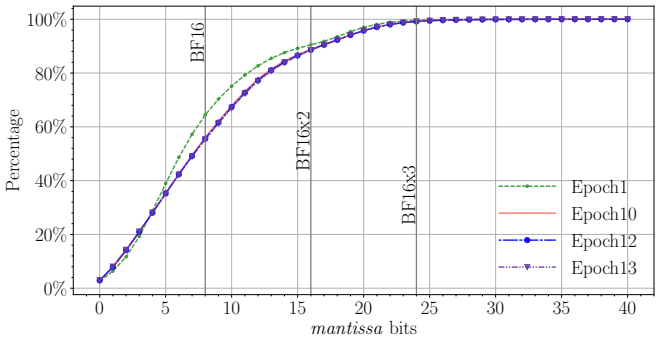


Fig. 15: *Swamping* analysis on LSTMx2 Model.

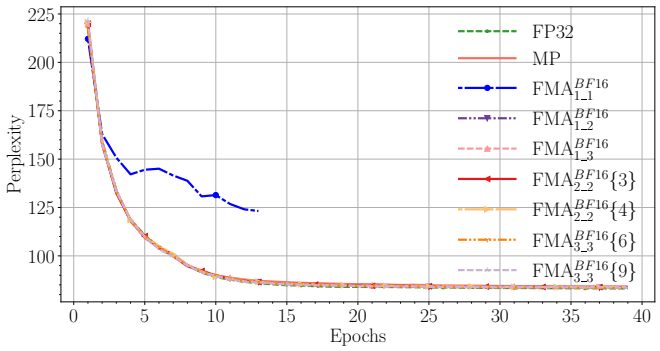


Fig. 14: LSTMx2 Test Perplexity on PTB dataset

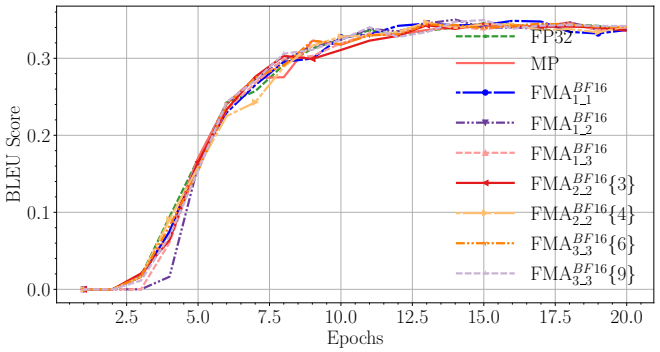


Fig. 16: Transformer BLEU Score on IWSLT16.

Task (NMT) using the IWSLT16 dataset. Figure 16 has the BLEU Score evolution doing the translation from Dutch to English. For this network, all of the approaches produce comparable results with respect to FP32. Transformer-based models display robust numerical properties, as $FMA_{1\_1}^{bf16}$ produces state-of-the-art results [36].

We consider an additional experiment involving NMT on Multi30K dataset using a transformer-based model. Again, all approaches obtain state-of-the-art levels of accuracy. Figure 17 shows that the models converge reaching the same training perplexity. We also compute the final BLEU Scores, which are 35.67, 36.05, 36.33 and 36.08 for FP32, $FMA_{1\_1}^{bf16}$, $FMA_{2\_2}^{bf16}$ {4} and $FMA_{3\_3}^{bf16}$ {6}, respectively. Another example that transformer-based models display robust numerical properties with BF16 low precision operators.

To explain the good performance of $FMA_{1\_1}^{bf16}$ with transformer models, we carry out the *swamping* analysis on several epochs when training IWSLT16. Figure 18 shows how $FMA_{1\_1}^{bf16}$ is enough to represent at least 70% of all FMA calculations without *swamping*. In comparison, for the LSTMx2 network this number is just 55%.

In conclusion, $FMA_{n\_m}^{bf16}$ operators match the accuracy of

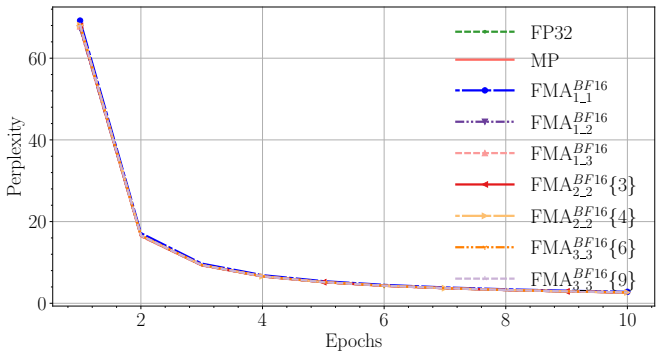


Fig. 17: Transformer Training Perplexity on Multi30k.

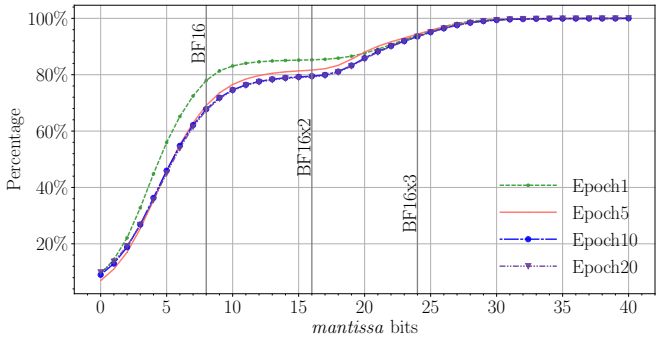


Fig. 18: *Swamping* analysis on Transformer model using IWSLT16 dataset.

FP32 when training NLP models without the need of FP32 arithmetic. The best suited operators are FMA$^{bf16}_{1_2}$ and the two FMA$^{bf16}_{2_2}$ variants, as FMA$^{bf16}_{1_1}$ fails to converge on the LSTMx2 model.

7.3 Training Accuracy: Recommender Systems

Figure 19 shows the training process of the recommender system introduced in Section 6.3. The y-axis shows the Normal Discounted Cumulative Gain (NDCG), which is a metric to quantify the accuracy of the recommender system. As in most previous models, FMA$^{bf16}_{1_1}$ fails to deliver competitive results. FP32 and MP obtain the same score of 66.76%, and the rest of the approaches deliver similar results that are within $\pm 0.2\%$. The best result is obtained by FMA$^{bf16}_{2_2}$ {4} with 66.95%. We can again conclude that FMA$^{bf16}_{n_m}$ operators can deliver state-of-the-art accuracy.

7.4 Performance Evaluation

We use the Sniper simulator to evaluate the performance benefits of the FMA$^{bf16}_{1_1}$, FMA$^{bf16}_{1_2}$, and FMA$^{bf16}_{2_2}$ {4} operators. Table 6 shows the performance gains we have when training ResNet101 (CIFAR10), the transformer model (Multi30k), and LSTMx2 (PTB) with respect to an FP32 baseline. The obtained speed-ups in ResNet101 are 1.35×, 1.34× and 1.28× for FMA$^{bf16}_{1_1}$, FMA$^{bf16}_{1_2}$, and FMA$^{bf16}_{2_2}$ {4} respectively. The transformer model presents similar results, while LSTMx2 shows a slightly reduction on the performance improvements due to an instruction mix with a lower FMA instruction count.

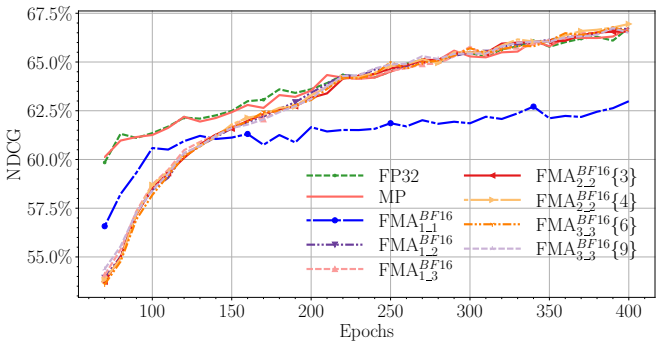


Fig. 19: Normal Discounted Cumulative Gain on MovieLens Dataset.

TABLE 6: Performance speed-up estimations using the Sniper Simulator

Model	One Batch			
	FP32	FMA$^{bf16}_{1_1}$	FMA$^{bf16}_{1_2}$	FMA$^{bf16}_{2_2}$ {4}
ResNet101	1.00×	1.35×	1.34×	1.28×
Transformer (Multi30k)	1.00×	1.37×	1.35×	1.29×
LSTMx2	1.00×	1.31×	1.30×	1.22×

Performance gains stem from the additional throughput provided by the wider FMA functional units. However, our micro-architectural simulations consider all executed instructions, not just FMAs. The larger the percentage of FMA instructions is with respect to the total instruction count, the more potential FMA$^{bf16}_{n_m}$ operators have in terms of performance improvements. For example, the instruction mix of ResNet101 has 57.6% non-floating point instructions (i. e. 42.4% are FP), and 39.5% of the total instruction count are FMAs. The FMA$^{bf16}_{1_2}$ operator accelerates FMA instructions by 2.80× with respect to FP32, which leads to the reported 1.34× for the whole execution. FMA$^{bf16}_{n_m}$ provides larger floating-point throughput than FP32 units. Therefore, memory- or software-level optimizations to feed these compute units more efficiently could provide additional benefits.

These results demonstrate that FMA$^{bf16}_{n_m}$ operators not only achieve state-of-the-art accuracy, but also deliver substantial performance improvements with respect to FP32 functional units for the same area budgets.

8 Related Work

As the complexity of new DNN models increases [37], training these models translates into large computational and environmental costs [10], valued at millions of dollars. Multiple proposals try to reduce these costs by using reduced precision strategies in order to take advantage of the favorable area and power trade-offs associated with narrower hardware units [3].

Several studies show that reduced precision techniques [3], [8] diminish computing time and energy consumption when training DNNs. Micikevicius et.al [7] proposes an MP technique using FP16 on Nvidia GPUs while Kalamkar et.al [6] use BF16; however, both require FP32 arithmetic to compute critical sections such as WU and BN layers.

Additionally, Graphcore [38] presents a hardware accelerator that also targets MP training with FP16 datatypes. They also define a new numerical datatype called AI-Float, which uses stochastic-rounding hardware to keep accuracy across models. While these methods achieve comparable accuracy with respect to FP32 training in their evaluations, they rely on FP32 computations in critical layers and accumulators to aid training convergence on deep networks.

Sun et.al [8] propose a complex methodology to enable ultra-low precision training using a 4-bits numerical format. This proposal delivers worse precision than the state-of-the-art and requires additional steps, such as: (i) a new GradScale technique to adjust the gradients to the FP4 range on a layer-by-layer basis, and (ii) a hybrid approach to change to FP8 in some cases where FP4 fails to converge. While the gains can be large, this approach requires significant hardware and software modifications to existing platforms and frameworks, and does not avoid the need of 32-bits accumulations. Moreover, this approach requires an ad-hoc recipe to train each network, which severely undermines its generality and applicability.

Fu et.al [9], [10] propose a dynamic precision training approach that cyclically employs between three and eight bits of precision. Again, such a methodology requires ad-hoc hardware and additional steps, such as: (i) to compute the lower bound of precision at the beginning of the training, and (ii) a scheduling method used during the training process to select the numerical formats. This approach requires WU calculations to use full FP32 FMAs during DNN training, and all FMA accumulators use FP32. Moreover, these works do not compare directly against FP32 but to other low precision schemes as [39], [40].

In contrast, $\text{FMA}_{\text{n\_m}}^{\text{bf16}}$ relies on the already well-known and widely adopted BF16 datatype to offer different precision tiers. Additionally, applying $\text{FMA}_{\text{n\_m}}^{\text{bf16}}$ does not require additional ad-hoc steps that vary depending on the DNN model. $\text{FMA}_{\text{n\_m}}^{\text{bf16}}$ can be deployed in the context of low-end computing systems as it just requires a small and cheap BF16 FMA unit. $\text{FMA}_{\text{n\_m}}^{\text{bf16}}$ provides different accuracy levels and $\text{FMA}_{\text{1\_2}}^{\text{bf16}}$ or $\text{FMA}_{\text{2\_2}}^{\text{bf16}}$ arithmetic can be used when $\text{FMA}_{\text{1\_1}}^{\text{bf16}}$ is not enough, thereby lowering training costs with respect to conventional FP32 arithmetic while delivering state-of-the-art accuracy.

9 Conclusion and Future Work

In this paper we propose a new class of FMA operators, $\text{FMA}_{\text{n\_m}}^{\text{bf16}}$, that rely entirely on BF16 arithmetic but can achieve FP32 accuracy through compound datatypes (BF16xN). We analyze the suitability of these BF16xN datatypes for DNN training and find that they are able to significantly mitigate the representation errors and *swamping* issues commonly observed when using a single BF16 literal. We then define and characterize seven $\text{FMA}_{\text{n\_m}}^{\text{bf16}}$ operators that feature different levels of accuracy and theoretical throughput improvements at iso-area with respect to an FP32 FMA unit.

To evaluate training accuracy of the proposed operators on a wide range of DNN workloads we develop SERP, a binary analysis tool that enables $\text{FMA}_{\text{n\_m}}^{\text{bf16}}$ emulation and seamlessly works with PyTorch, Caffe, or Tensorflow. For all the evaluated networks, the proposed $\text{FMA}_{\text{2\_2}}^{\text{bf16}}$ $\{4\}$ or $\text{FMA}_{\text{1\_2}}^{\text{bf16}}$ operators obtain comparable FP32 state-of-the-art accuracy. Demonstrating that it is possible to train DNNs exclusively with BF16 arithmetic for all layers. In addition, we evaluate $\text{FMA}_{\text{n\_m}}^{\text{bf16}}$ in terms performance speed-ups using micro-architectural simulations. We show that $\text{FMA}_{\text{1\_1}}^{\text{bf16}}$, $\text{FMA}_{\text{1\_2}}^{\text{bf16}}$, and $\text{FMA}_{\text{2\_2}}^{\text{bf16}}$ $\{4\}$ operators achieve speed-ups of $1.35\times$, $1.34\times$ and $1.28\times$ on ResNet101, respectively, when compared to FP32 at equivalent area.

As future work we plan to exploit the different $\text{FMA}_{\text{n\_m}}^{\text{bf16}}$ accuracy levels by mapping the different formats to layers depending on the precision demands of each layer. Finally, a dynamic approach that switches between two or more $\text{FMA}_{\text{n\_m}}^{\text{bf16}}$ operators can help increase precision while using the cheapest FMA operator ($\text{FMA}_{\text{1\_1}}^{\text{bf16}}$) for a large portion of the computations.

Acknowledgments

Marc Casas has been partially supported by the Grant RYC-2017-23269 funded by MCIN/AEI/10.13039/501100011033 and by ESF Investing in your future. Adria Armejach is a Serra Hunter Fellow and has been partially supported by the Grant IJCI-2017-33945 funded by MCIN/AEI/10.13039/501100011033. John Osorio has been partially supported by the Grant PRE2019-090406 funded by MCIN/AEI/10.13039/501100011033 and by ESF Investing in your future. This work has been partially supported by Intel under the BSC-Intel collaboration and European Union Horizon 2020 research and innovation programme under grant agreement No 955606 - DEEP-SEA EU project.

References

[1] A. Tiwari, G. Trivedi, and P. Guha, "Design of a low power bfloat16 pipelined mac unit for deep neural network applications," in *2021 IEEE Region 10 Symposium (TENSYMP)*, 2021, pp. 1–8.

[2] G. Henry, P. T. P. Tang, and A. Heinecke, "Leveraging the bfloat16 artificial intelligence datatype for higher-precision computations," 2019.

[3] J. Y. F. Tong, D. Nagle, and R. A. Rutenbar, "Reducing power by optimizing the necessary precision/range of floating-point arithmetic," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 8, no. 3, pp. 273–286, 2000.

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

[5] Nvidia. (2021) Improve tensor core operations. [Online]. Available: https://bit.ly/3s6vkzI

[6] 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.

[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] X. Sun, N. Wang, C.-Y. Chen, J.-M. N. Ankur, A. Xiaodong, C. Swagath, V. Kaoutar, E. Maghraoui, V. Srinivasan, and K. Gopalakrishnan, "Ultra-Low Precision 4-bit Training of Deep Neural Networks," in *Neurips*, 2020.

[9] Y. Fu, H. You, Y. Zhao, Y. Wang, C. Li, K. Gopalakrishnan, Z. Wang, and Y. Lin, "Fractrain: Fractionally squeezing bit savings both temporally and spatially for efficient dnn training," in *Neurips*, 2020.

[10] Y. Fu, H. Guo, M. Li, X. Yang, Y. Ding, V. Chandra, and Y. Lin, "{CPT}: Efficient deep neural network training via cyclic precision," in *International Conference on Learning Representations*, 2021. [Online]. Available: https://openreview.net/forum?id=87ZwsaQNHPZ

[11] 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.
[12] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," *CoRR*, 2015.
[13] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "Mobilenetv2: Inverted residuals and linear bottlenecks," 2019.
[14] W. Zaremba, I. Sutskever, and O. Vinyals, "Recurrent neural network regularization," 2015.
[15] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," 2017.
[16] 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.
[17] B. Trevett. (2020, jan) Neural machine translation by jointly learning to align and translate. [Online]. Available: https://bit.ly/38lD6v4
[18] X. Wang, X. He, M. Wang, F. Feng, and T.-S. Chua, "Neural graph collaborative filtering," in *Proceedings of the 42nd International ACM SIGIR Conference on Research and Development in Information Retrieval*. ACM, jul 2019. [Online]. Available: https://doi.org/10.1145%2F3331184.3331267
[19] Apr 1998. [Online]. Available: https://grouplens.org/datasets/movielens/100k/
[20] T. E. Carlson, W. Heirman, and L. Eeckhout, "Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulations," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, Nov. 2011, pp. 52:1–52:12.
[21] Nvidia. Nvidia tensor cores. [Online]. Available: https://www.nvidia.com/en-us/data-center/tensorcore/
[22] S. Wang and P. Kanwar, "Bfloat16: The secret to high performance on cloud tpus," 2019. [Online]. Available: https://bit.ly/2SlSYqq
[23] Intel. (2020) Intel architecture instruction set extensions and future features programming reference. [Online]. Available: https://intel.ly/3czWOpS
[24] Nvidia. (2020, may) Nvidia a100 tensor core gpu architecture. [Online]. Available: https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/nvidia-ampere-architecture-whitepaper.pdf
[25] 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 Parallel Processing - International Conference*, 2019.
[26] Intel. (2020) Intel math kernel library. [Online]. Available: https://software.intel.com/en-us/mkl
[27] A. Krizhevsky, "Learning Multiple Layers of Features from Tiny Images," Tech. Rep., 2009.
[28] A. U. Durmus, "Replication of recurrent neural network regularization by zaremba," Sep 2019. [Online]. Available: https://github.com/ahmetumutdurmus/zaremba/
[29] A. Gordic, "Original pytorch transformer model," Oct 2020. [Online]. Available: https://github.com/gordicaleksa/pytorch-original-transformer
[30] T. Ben, "Pytorch seq2seq," Jul 2018. [Online]. Available: https://github.com/bentrevett/pytorch-seq2seq
[31] Metahexane, "Neural graph collaborative filtering algorithm in pytorch." [Online]. Available: https://github.com/metahexane/ngcf_pytorch_g61
[32] A. Paszke, S. Gross, S. Chintala, and G. Chanan. (2020) Pytorch. [Online]. Available: https://github.com/pytorch/pytorch
[33] Intel. Intel deep neural network library. [Online]. Available: https://github.com/intel/mkl-dnn
[34] ——, "Intel architecture instruction set extensions programming reference," Dec 2020. [Online]. Available: https://intel.ly/3j9tCsM
[35] K. Audhkhasi, O. Osoba, and B. Kosko, "Noise benefits in back-propagation and deep bidirectional pre-training," in *IJCNN*, 2013.
[36] T. Tambe, E.-Y. Yang, Z. Wan, Y. Deng, V. J. Reddi, A. M. Rush, D. Brooks, and G.-Y. Wei, "Adaptivfloat: A floating-point based data type for resilient deep learning inference," vol. arXiv, no. 1909.13271, 2019. [Online]. Available: https://arxiv.org/pdf/1909.13271.pdf
[37] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," 2020.
[38] N. Toon, "Introducing 2nd generation ipu systems for ai at scale," Jul 2020. [Online]. Available: https://shorturl.at/awxyB
[39] S. Zhou, Y. Wu, Z. Ni, X. Zhou, H. Wen, and Y. Zou, "Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients," 2018.
[40] Y. Yang, S. Wu, L. Deng, T. Yan, Y. Xie, and G. Li, "Training high-performance and large-scale deep neural networks with full 8-bit integers," 2019.

John Osorio Is a Ph.D. student at Barcelona Supercomputing Center since 2018 in collaboration with Intel. John got his bachelor's and master's degree in Computer Science at Universidad Tecnologica de Pereira (Colombia) in 2016. Currently, he is working on the interaction between machine learning and computer architecture, specifically reducing numerical data type precision to train deep neural networks.

Adrià Armejach Is a Serra Húnter Fellow at Universitat Politècnica de Catalunya (UPC) and an associate researcher at the Barcelona Supercomputing Center (BSC). His interests include computer architecture, parallel computing, memory systems, and performance evaluation. He received his Ph.D. from UPC in 2014. Since then he has led the technical contributions of multiple FP7 and H2020 projects, working on emerging memory technologies, heterogeneous architectures, simulation methodologies for next-generation large-scale HPC machines, and vector architectures.

Eric Petit joined Intel in 2016, he is now part of the Math Library Pathfinding group. He received his Ph.D. from University of Rennes at INRIA on compiler technology for GPGPU in 2009. He spend 2 years at University of Perpignan and 6 years at University of Versailles where he led a team participating in multiple European project on performance analysis tools and parallel runtimes and programming, with multiple open-source software contributions and peer-reviewed publications. His current research interests are on computer arithmetic, floating point precision and accuracy analysis and optimization.

Greg Henry Is a senior principal engineer at Intel Corporation. He got his Ph.D. in Applied Mathematics from Cornell University under Prof. Charles Van Loan and joined Intel in 1993. His research areas are in linear algebra and machine learning and helped run all software development at the tail end of the Intel Supercomputing division. He then became the Intel(R) Math Kernel Library Architect and now works on math library research in a Pathfinding group. He lives in the Pacific Northwest of the United States. Outside of work, he has a fourth-degree black belt and loves to write.

Marc Casas Marc Casas is a senior researcher at the Barcelona Supercomputing Center (BSC) and lecturer at the Universitat Politècnica de Catalunya (UPC). His research lays between computer architecture (e.g. memory address translation, vector architectures) and high-performance computing (e.g. sparse linear algebra, parallel deep learning). He is the technical lead of the SONAR (SOftware Research vehicles for New ARchitectures) research group, composed of PhD students, engineers, and post-docs. Marc has lead BSC contributions to several european projects and research collaborations with Intel and IBM.