FASE: A Fast, Accurate and Seamless Emulator for Custom Numerical Formats

John Osorio ✉[1,2], Adrià Armejach[1,2],
Eric Petit[3], Greg Henry[3], and Marc Casas[1,2]

[1] Universidad Politecnica de Catalunya, Barcelona, Spain
[2] Barcelona Supercomputing Center, Barcelona, Spain
[3] Intel Corporation, Oregon, USA
john.osorio@bsc.es

Abstract. Deep Neural Networks (DNNs) have become ubiquitous in a wide range of application domains. Despite their success, training DNNs is an expensive task which has motivated the use of reduced numerical precision formats to improve performance and reduce power consumption. Emulation techniques are a good fit to understand the properties of new numerical formats on a particular workload. However, current state-of-the-art techniques are not able to perform this tasks quickly and accurately on a wide variety of workloads.
We propose FASE, a Fast, Accurate and Seamless Emulator that leverages dynamic binary translation to enable emulation of arbitrary numerical formats. FASE is *fast*; allowing emulation of large unmodified workloads, *accurate*; emulating at instruction operand level, and *seamless*; as it does not require any code modifications and works on any application or DNN framework without any language, compiler or source code access restrictions. We evaluate FASE using a wide variety of DNN frameworks and large-scale workloads. Our evaluation demonstrates that FASE achieves better accuracy than coarser-grain state-of-the-art approaches, and shows that it is able to evaluate the fidelity of multiple numerical formats and extract conclusions on their applicability.

Keywords: Numerical emulation · Reduced precision · DNN training.

1 Introduction

Current trends on DNNs indicate that training costs will continue to grow as state-of-the-art DNNs feature increasingly large parameter counts [3]. There are already approaches on reducing the training computation costs via mechanisms that incur accuracy degradations [16, 29, 32]. Additionally, there are approaches able to reduce training costs without reducing DNNs accuracy. These approaches rely on reduced computer number formats [10, 11, 33, 34]. To decide among all potential format designs which ones display the best opportunities for efficient and accurate DNN training, it is critical to empirically evaluate them with as much fidelity as possible and on as many real neural net topologies and real

input datasets as possible. The emulation of these reduced precision approaches becomes one of the most important and costly phases to evaluate the reliability of new numerical data types. The emulation helps to avoid cost overrun, by avoiding costly hardware implementations.

TensorQuant [26], proposes two source level approaches, intrinsic and extrinsic, to emulate low precision using Tensorflow. The extrinsic approach is an approximation where the rounding process is done just on high level operators like convolutions. This is the mode implemented in QPyTorch [39] to address the PyTorch framework. The intrinsic approach rounds each individual floating point operation and displays a latency of 50× with respect to native executions. It is a source level approach that can be used to evaluate all implementation of neural network based on Tensorflow. All of these approaches are designed targeting specific DNN frameworks and require changes on the framework and model source code. Other tools like Verificarlo [4] work at the compiler level, and can be applied to any Python framework; but, they do require complex recompilation.

To overcome these issues we propose FASE: a fast, accurate and seamless tool that enables the emulation of custom numerical formats on any application. FASE relies on dynamic binary instrumentation using PIN [27] to perform fine-grain instruction-level instrumentation. In addition, FASE seamlessly works on any application or DNN framework without any language, compiler or source access restrictions. Since no code modification or recompilation steps are necessary, FASE guarantees that the instrumented binary matches the original one. Therefore, FASE works on all DNN frameworks, such as: Caffe [17], Tensorflow [1] and PyTorch [30]. While fine-grain instrumentation can inject large latencies, we propose a set of optimizations that enable FASE to emulate unmodified applications on large input sets with latencies that range from 17× to 39×, which are comparable to other fine-grain state-of-the-art techniques. As a result, FASE enables hardware architects to understand numerical behaviour before committing to costly hardware implementations. This paper makes the following contributions:

- We propose FASE[4], an emulation tool for custom numerical formats that enables accurate emulation of large workloads without requiring any source code modifications or access to third-party dynamically linked libraries.
- We design performance optimizations that enable accurate emulation with low overhead to support large-scale experimentation.
- An exhaustive evaluation campaign that demonstrates that FASE achieves better accuracy with respect to other state-of-the-art coarser-grain approaches, as well as large-scale experiments using multiple numerical formats that demonstrate that FASE is able to evaluate the fidelity of numerical formats.

2 Background and Motivation

The increasing demand for computing power in machine learning training motivates the use of reduced numerical precision formats. It has lead to a myr-

[4] Source code is publicly available at https://gitlab.bsc.es/josorio/fase

Table 1: Comparison of state-of-the-art proposals.

Features	**Emulators**				
	RPE [7]	**QPyTorch [39]**	**TensorQuant [26]**	**Verificarlo [4]**	**FASE**
Fast	✗	✓✓	✓	✓✓	✓
Accurate	✓	✗	✓	✓	✓
Seamless	✗	✗	✗	✗(recompilation)	✓
Dynamic Libraries	✗	✗	✗	✗(Lib. recompilation)	✓
Independent	✗	✗	✓	✗(compiler dep.)	✓

iad of proposals for custom reduced precision numerical formats, both floating-point and integer, to improve the large computational and energy costs of training DNN. These workloads can tolerate well low-precision formats in certain computations, with proposals that go as low as 4-bit numerical representations [10, 11, 34].

Machine learning and DNN workloads in particular heavily rely on linear algebra kernels that can greatly benefit from low-precision formats in order to reduce memory bandwidth and storage usage, as well as improve compute throughput by leveraging vectorisation or accelerators that can fit more elements per instruction. An example is the adoption of the new Brain Float 16-bit (BF16) numerical format, extensively used in DNN workloads, by most hardware vendors [2, 28, 31]; which may be used to substitute the IEEE 754 32-bit floating-point typically employed.

In order to evaluate new numerical formats without available hardware support, several tools and methodologies to emulate low-precision numerical formats have been proposed. Table 1 qualitatively compares multiple state-of-the-art proposals on a number of key features. We consider a proposal is *fast* if it is feasible to emulate unmodified applications on large input sets, i.e., if the workload does not need to be scaled down to have feasible emulation times. *Accurate* means that the emulation is done at a fine-grain granularity (e.g., per instruction), rather than at coarse-grain granularity (e.g., per function) which may lead to results that are more accurate than actual computations at low precision. *Seamless* means that the emulated code does not need to be modified while *dynamic libraries* means that the tool is able to emulate code from dynamically linked libraries which may not always be open source. Finally, *Independent* is for tools that can work on any programming language and are also compiler independent.

The Reduced Precision Emulator (`rpe`) [7] is an emulator which supports reduced precision that can be computed on the available hardware format and rounding. The tool operates in a fine-grain manner. They report overheads from 10-70× on small emulated workloads. However, like all other source level approaches, code modification interfere with compiler optimizations impacting numerical accuracy [8]. Furthermore, it is currently restricted to Fortran applications. `Verificarlo-Vprec` [4, 8] propose an LLVM compiler pass, at the end of

the optimization passes, replacing all floating-point operations by user defined ones. Vprec enables emulating reduced precision formats like BF16. It allows accurate per operation rounding, with a latency from 3 to 17× according to their experiments [4]. It handles all programming languages supported by LLVM. However it does require the recompilation of all the application and its static and dynamic dependencies. Which is a tedious process, and not always possible for closed source libraries. However, they support Python environment by proposing a prebuild linux docker image.

There are two main tools that focus specifically on DNN workloads, TensorQuant [26] and QPyTorch [39]. TensorQuant is a quantization toolbox for the Tensorflow framework that provides multiple methods to apply reduced precision formats. They propose a coarse-grain method that applies the rounding processes at the end of each DNN layer, all intermediate computations inside a layer are not altered. TensorQuant also has a fine-grain operation-by-operation method that enables accurate emulation with a reported latency increase of around 20×. Using the fine-grain method is complex, as the user needs to re-implement each composite operation using C++ calls. It only works on Tensorflow, requires code modifications on each workload and does not instrument dynamically linked libraries. Some low-precision DNN training schemes like [10,11] use QPyTorch [39] as reduced precision framework. QPyTorch is a fast reduced-precision emulation framework for PyTorch. QPyTorch first represents the low precision numbers as their corresponding floating point number, then operates using single-precision floating point computation and then removes the extra precision through a final quantization step. While it is a fast methodology, the reduced-precision transformations are done at coarse-grain level; it may not capture the real effects of using reduced precision; it requires code modifications on each workload; it does not instrument dynamically linked libraries.

In contrast, FASE seamlessly works on any ML framework and is able to emulate code in dynamically linked libraries. This is crucial in DNN training workloads as most low level compute kernels are implemented in highly optimized external libraries. In addition, we make FASE accurate by operating at fine-grain. To reduce the latency of having accurate emulation we implement multiple optimizations that enable FASE to emulate large workloads with overheads that are competitive with other state-of-the-art proposals. We detail our design choices in Section 3, the implementation and performance optimization in Section 4, the strategy we apply to evaluate the tool on machine learning frameworks in Section 5 and evaluate accuracy and performance in Section 6.

3 FASE Design

Our goal is to design FASE with simplicity in mind by enabling fast, accurate and seamless emulation of reduced precision formats. In addition, we want our tool to be able to emulate code of external dynamically linked libraries, as many applications rely on such libraries which contain key optimized routines.

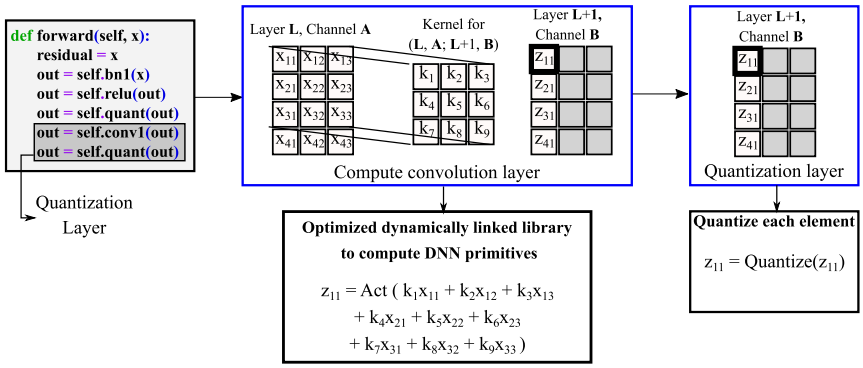

Fig. 1: Steps for coarse-grain emulation on a convolutional layer.

Figure 1 shows a forward pass example to demonstrate the operations of extrinsic coarse-grain reduced precision emulation. In this example, a convolution layer performs a dot product using a 3x3 filter to compute each element of the output layer *L+1*. These low level compute kernel implementations are typically found in optimized external libraries such as Intel oneDNN [19]. On the left side, the application needs to be modified to indicate where the conversion (quantization) takes place prior to the kernel. After the output layer *L+1* computation, a quantization and rounding step is performed over each element to obtain the desired reduced precision representation. This is a simple and fast methodology that allows to use well-known optimized libraries to compute the convolution. However, this method is not accurate as all operations within the layer employ the original single-precision format, leading to optimistic results not as accurate than using a fine-grain approach or real hardware.

FASE aims to provide an accurate and seamless method. To achieve this fine grain emulation, we propose to leave the target application unmodified and operate at binary level intercepting the executed machine instruction. By identifying key floating-point instructions, for which we can modify the input and output operands, FASE can seamlessly work on any application and DNN framework including dynamically linked external libraries.

4 FASE Implementation

4.1 Overview

In order to provide a fast, accurate and seamless experience; FASE relies on Dynamic Binary Translation (DBT). DBT enables modifications in the dynamic instruction flow of any application binary, as well as on any dynamically linked libraries the binary invokes. These modifications are done during the *instrumentation* step, which is executed only once.

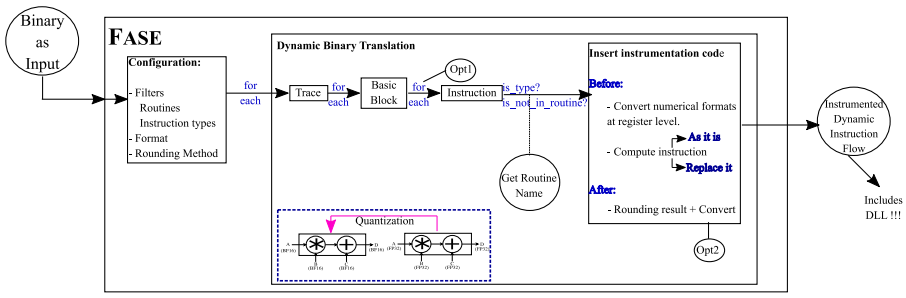

Fig. 2: FASE Implementation Overview

Figure 2 shows an overview of the DBT instrumentation step on FASE. FASE can be attached to any binary, and is configured through a simple configuration file that specifies the desired instrumentation parameters in terms of routines and instructions to be instrumented as well as the emulated reduced precision format and rounding method. The DBT step which performs the instrumentation goes through each statically defined basic block once, and for each instruction it can insert instrumentation code. In our context, for each instruction of interest, we want to perform up to three code insertions:

1. **Before:** Insert code that converts the source registers of the instruction to the desired reduced precision format and applies the desired rounding.
2. **Instruction:** In most cases the instruction can be executed as is with the modified source registers. In some 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 employing compound data types or custom formats that cannot be represented with the original numeric representation.
3. **After:** Insert code that converts the output to the desired reduced precision format and applies the rounding mechanism.

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 any external libraries, is executed. Analysis is the most compute expensive step as the modified instruction flow with code insertions is executed.

4.2 Features and Configuration Options

FASE has a number of built-in features and configuration options that simplify the use of the tool and enable fine tuning of the emulation process.

Filters: There are two main types of filters: routine names and instruction types. Users can specify routines that should not be instrumented, i.e., routines that require high precision or that are not of interest for the target application. In terms of instruction types, FASE provides easy tags to identify most types,

Listing 1.1: Basic block optimization on a ResNet50 basic block. Only operands in bold need to be converted. Underlined operands are source and destination and need to be converted twice.

```
vfmadd213ps zmm4, zmm2, zmmword ptr [rax+r9*4]
vfmadd231ps zmm5, zmm4, zmm3
vfmadd231ps zmm6, zmm5, zmm0
vfmadd213ps zmm7, zmm2, zmmword ptr [rax+r9*4+0x20]
vfmadd231ps zmm8, zmm7, zmm3
vfmadd231ps zmm9, zmm8, zmm0
```

for example, all floating-point instructions or just specific instruction types like FMAs. Different instruction types can be defined to use different reduced precision numerical formats or rounding methods.

Dynamically changing precision during analysis step: FASE supports an inter-process communication (IPC) method that enables signaling FASE from the emulated application to dynamically change emulation behaviour. This does require modifications to the emulated application, in the form of simple function calls, to signal FASE to change its operation mode.

Numerical formats and rounding methods: FASE can support any custom low precision numerical format and rounding method. If the format is compatible with the original instruction binary size of exponent and mantissa, then the inserted code in the instrumentation phase is simpler, as it just has to convert the source and destination registers. If the format cannot be operated by the original instruction, it is replaced by code that can perform the operation.

4.3 Optimizations

In this section we explain the different optimizations we apply to FASE to match state-of-the-art proposals while achieving high emulation accuracy. For all optimizations, FASE is performing as much work as possible in the instrumentation step to lower analysis overheads, as instrumentation is performed only once statically per basic block. Therefore, we apply all the filters during the instrumentation step and only insert the necessary code for the selected instructions and routines, which will run in the analysis step.

We started from a straight forward implementation where each FP instruction is instrumented and the computation in the analysis phase in FASE is not optimized. This `unopt` version will be our upper bound for performance against which the following optimization will be evaluated in Section 6. On the other end, the fully optimized version will be referred as `full opt`.

Basic-block level optimization: During the instrumentation step we perform a basic-block level optimization that enables a substantial reduction of inserted code. We keep track of all source and destination register names that will be converted and rounded, if one of these registers is used as source in a subsequent instruction within the basic-block, it is safe to skip the conversion

Listing 1.2: Vectorized BF16 with RNE conversion

```
inline __m512 ToBFloatRNEVec (__m512* input)
{
    __m512i MSB_mask = _mm512_set1_epi32(0x80000000);
    __m512i LSB_mask = _mm512_set1_epi32(1);
    __m512i mask = _mm512_set1_epi32(0xFFFF0000);
    __m512i qnan_mask = _mm512_set1_epi32(0x7FC00000);
    __m512i rounding_mask = _mm512_set1_epi32(0x7FFF);

    __m512i tmp = _mm512_srli_epi32(*(__m512i*)input, 16);
    tmp = _mm512_and_si512(tmp, LSB_mask);
    __m512i rounding_bias = _mm512_add_epi32(tmp, rounding_mask);

    __m512i MSB_set = _mm512_and_si512(*(__m512i*)input, MSB_mask);

    tmp = _mm512_xor_si512(*(__m512i*)input, MSB_set);
    tmp = _mm512_add_epi32(tmp, rounding_bias);
    tmp = _mm512_or_si512(tmp, MSB_set);

    __mmask16 not_nan_mask = _mm512_cmp_ps_mask(*input, *input, _CMP_EQ_OQ);

    tmp = _mm512_mask_and_epi32(qnan_mask, not_nan_mask, tmp, mask);

    *input = *(__m512*)&tmp;

    return *input;
}
```

and rounding of that register as it is already in the desired target numerical format. Since it is quite common for destination and source registers to be reused in subsequent instructions, this optimization is very effective at reducing the overheads during the analysis step, as no work needs to be done for many source operands. Listing 1.1 shows an example of the traces generated by DNN frameworks. Only the highlighted operands need to be converted (underlined need to be converted twice as they are source and destination registers), saving 29.2% of the time in this particular basic block. In Figure 2 and Section 6.2 we refer to this optimization as `Opt1`.

Vectorization: When instrumenting vectorized code, which is common in HPC and DNN low-level optimized kernels, FASE has support to do the numerical conversions and rounding methods also in a vectorized manner. This optimization greatly reduces the latency of instrumented vector instructions.

Listing 1.2 presents the vectorized optimization FASE implements to boost the performance, reducing the emulation latency as Section 6.2 shows. In this example, we implement the rounding process using AVX512 Intel Intrinsics, but 256bit, 128bit and scalar implementations are also available. This allows us to

round the elements in the AVX vector register in a data parallel manner. Lines 3-7 define the whole set of masks we need to do the rounding. Lines 9-11 compute the rounding bias. Then, we need to do an unsigned integer addition between the rounding bias and the input, however AVX512 does not support it. Due to this issue FASE uses a few additional instructions to achieve it: we save the MSB bits of each element of the AVX512 vector (line 13), then we set all MSB of the input to zero in line 15, then compute a signed integer addition (line 16) and finally reset the MSB bits to its original value in line 17. Finally, FASE just needs to check for NaN values and return the AVX512 vector. In Figure 2 and 6.2 section we refer to this optimization as `Opt2`.

5 Applying FASE to DNN Training Workloads

In this paper, the main use case for FASE is its applicability to DNN training workloads. These workloads have high computational cost while tolerating reduced precision formats that FASE can emulate accurately. Multiple proposals to employ reduced precision training methodologies for DNN workloads exist. Some are based on emulation [21], while others target existing hardware [28].

5.1 Reduced precision formats

The need for reduced precision formats for DNN training has lead to numerous proposals. The most prominent to date, which is being adopted by most hardware vendors, BF16 format. BF16 retains the same dynamic range as FP32 as it has the same number of exponent bits (8), but has a shorter mantissa of just 7 bits. The use of a 16-bit format can alleviate memory storage and bandwidth requirements as well as increase computational throughput.

With FASE we can emulate multiple numerical formats to understand the behaviour of DNN training. For example:

- **Floating-point and integer formats:** FASE can easily support emulation of BF16, FP8, or other FP layouts by converting the necessary source and destination registers of floating-point instructions to these formats. Similarly, integer formats such as INT8 can also be emulated.
- **Compound numerical formats:** Compound datatypes based on the BF16 format have been proposed recently [14]. These formats link several (two or three) BF16 literals to increase precision while just operating using BF16 arithmetic. With FASE we can also emulate the use of these compound datatypes, as it is possible to change the semantics of the instrumented instruction to perform the necessary computation required. However the final result cannot always be stored in memory with the compound datatype and must be converted to the original type. This could be alleviated by using a shadow memory mechanism in future works [6].

5.2 DNN training strategies

FASE enables the implementation of popular DNN training approaches as well as experimenting with new methodologies.

Static strategies: For example, one can test the accuracy of a DNN model training when using BF16, FP8 or any other FP representation on the entire workload. Or emulate the already proposed mixed precision [21, 28] training technique, which is similar to using BF16 but uses the FP32 representation to do the accumulation step on FMA instructions.

Using routine filters: Certain functions (or DNN layers) require higher accuracy than others. For this reason FASE enables applying different numerical format conversions or avoid emulation altogether of certain routines. In DNN training, the *weight updates* and *batch normalization* layers are known to require FP32 precision to ensure network convergence. FASE enables this behaviour via simple configuration options.

Dynamic precision schemes and compound datatypes: FASE also enables to use of dynamic precision schemes that dynamically adapt to workload state at runtime. For example, it enables to adapt the numerical precision of the emulated format depending on how training convergence progresses in order to achieve the desired result.

6 FASE evaluation

Our experimental methodology considers the evaluation of FASE on several DNN frameworks. We consider: Caffe, Tensorflow, PyTorch and an additional test using the C programming language. Our experiments are performed on an Intel Xeon Platinum 8160 processors. We compile each framework from source enabling AVX512 Intel optimizations on all of them.

6.1 Emulation accuracy

Methodology: To evaluate emulation accuracy we use a common kernel present both in DNN training as well as a single precision matrix multiply (SGEMM). We implement this benchmark that multiplies two matrices using the Intel Math Kernel Library (oneMKL) [20]. We compiled the source code using GCC 8.1 with all the AVX512 optimizations active on the platform we use. We use as input two matrices: $A = 20000 \times 2000$ and $B = 2000 \times 10000$.

We execute this benchmark with regular FP32 precision to get the reference output. We then emulate the use of BF16 with RNE rounding using two approaches. Firstly, we apply quantization for each element of the output matrix to represent the numbers using BF16 and RNE rounding (*coarse-grain quantization* label) over the reference result. This is akin to the coarse-grain methods used by QPyTorch [39] and TensorQuant [26]. Secondly, we attach FASE on top of the benchmark binary, which instruments the code from the dynamically linked Intel MKL library. This enables to execute the workload using FASE

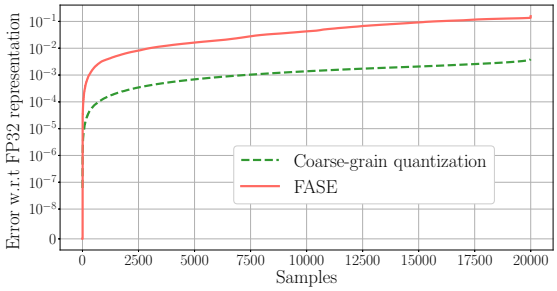

Fig. 3: Relative error of fine-grain and coarse-grain emulation methodologies (for BF16 and RNE rounding) with respect to a native FP32 execution.

fine-grain emulation at instruction level, representing both the input and output numbers in BF16 with RNE rounding (FASE label). Finally, we compare the relative error with respect to the FP32 reference of the two emulation strategies.

The following results illustrate that the fine-grain approach is a much more accurate approach to emulate reduced precision numerical formats.

Results: Figure 3 compares the relative error when employing fine-grain and coarse-grain emulation on the Intel MKL SGEMM kernel. The *x-axis* represents 20000 samples (elements) of the result matrix, sorted in terms of the absolute numerical error for the fine-grain and coarse-grain techniques. The *y-axis* displays the magnitude of the relative error with respect to the reference FP32 result. As can be seen in the figure, the relative error with FASE, which is close to what would be observed on a real hardware implementation, is consistently one order of magnitude higher than with the coarse-grain approach. Therefore, using the coarse-grain approach may lead to wrong assumptions about a particular reduced precision numerical format, as it delivers results that are more accurate than they should. A coarse-grain method cannot capture the errors that accumulate per instruction; however, FASE is able to track these errors and deliver a result that is much closer to reality.

In Section 6.3 we demonstrate FASE on full DNN training workloads and show that using BF16 exclusively fails to deliver state-of-the-art training accuracy for certain neural networks, demonstrating the importance of fine-grain accurate emulation of reduced precision formats.

6.2 FASE emulation overhead measurement

Methodology: To evaluate FASE latency overheads, and the impact of our optimizations, we propose an incremental evaluation process using the different FASE versions described in section 4.3. For all benchmarks, we compare each version against a reference native FP32 execution without instrumentation. Additionally, we report FASE's instrumentation overhead, which just increments a counter per instruction of interest, i.e, without computing any of the conversions or rounding processes. This instrumentation overhead allows us to get a lower

Table 2: FASE instrumentation latency and latencies for FASE unoptimized, after applying each optimization and fully optimized.

Workload (framework)	**FASE Instr.**	**Latency**			
		`Unopt`	`Opt1` Basic block	`Opt2` Vectorization	`Full Opt`
SGEMM (MKL)	15×	1809×	880×	82×	39×
ResNet50 (Caffe)	11×	1131×	553×	76×	30×
3DGan (Tensorflow)	7×	714×	340×	66×	28×
LSTM (PyTorch)	18×	1096×	551×	70×	29×
Transformer (PyTorch)	8×	818×	423×	36×	17×

bound of the tool overhead and estimate the cost of the conversion and rounding process in the fully optimized `full opt` version.

We evaluate each FASE version on several benchmarks. First we evaluate the SGEMM computation, as described in Section 6.1. Then we evaluate FASE on the following machine learning workloads:

- ResNet50 for one batch of size 64, with Intel-Caffe [17] 1.1.6a. We use Intel MKL-DNN [18] 0.18.0, and Intel MKL [20] 2019.0.3 to run the numerical kernels. To define and run the experiments we use the *pyCaffe* Python interface. The learning rate, gamma hyperparameter, momentum value, and weight decay are set to 0.05, 0.1, 0.9, and 0.0001, respectively.
- CERN 3DGAN [22] with Tensorflow [1] 1.15 and Keras [5] 2. We use the same MKL and MKLDNN libraries as in the ResNet50 case. The 3DGAN network is trained for one batch using the Adam optimizer and a batch size of 128. The training dataset consists of 180,000 25x25x25 three-dimensional images generated using HPC simulation for high-energy particles [22].

Finally, we consider two natural language processing models, for which we use a source-compiled version of PyTorch [30] version 1.8.0, Intel MKL-DNN [18] version 1.22.0, and Intel MKL library [20] version 2019.4.

- LSTMx2 model [38] on the PTB dataset. Following Zaremba et al. [38] we train one batch of the medium-sized model using the associated source code in [9] with a batch size of 20, an initial learning rate of 1, 2 LSTM layers, a hidden size of 650, a sequence length of 35, and a dropout value of 0.5.
- A transformer-based model [37] applied to the IWSLT16 dataset to translate between Dutch and English. We train for a batch size of 12000 using the Adam Optimizer with $\beta_1 = 0.9, \beta_2 = 0.98$, and $\epsilon = 10^{-9}$. We use the available code [12] and follow the author's additional instructions [12, 37].

Results: Table 2 shows the emulation latencies introduced by FASE when converting in a fine-grain manner the input and output operands to BF16 with

RNE rounding. We show the latency introduced by the *instrumentation* step of FASE in the "FASE Instrum." column, which on average is of 12×. This is the latency introduced just by counting the number of instructions of interest, and is therefore a lower bound of the overhead imposed by Intel Pin dynamic binary translation in FASE.

Regarding the latencies that include the emulation of the reduced precision format, we first show the latencies for an unoptimized version of FASE (Unopt.). This approach leads to latencies of up-to 1809×, which may deem the execution of large workloads unfeasible in a reasonable amount of time.

The *Basic-block* optimization, which refers to the *Opt1* version that avoids redundant rounding of registers, reduces FASE overhead by around half ranging from 340× to 880×. The observed latency reductions are inline with the amount of operands that need to be modified, as this optimization reduces by 50.89% the number of operands that FASE needs to convert in ResNet50.

The *Opt2* version in the *Vectorization* column is measuring the improvement we propose with custom AVX512 conversion and rounding process at analysis level using Intel Intrinsics. It results in a substantial speed up reducing FASE overhead latencies to the 36× to 82× range, emphasizing the importance of vectorizing the code on modern wide vector architectures.

Finally, we apply both the basic block *Opt1* and vectorization *Opt2* optimizations to our final *Full Opt* FASE version. It further reduces the final overhead down to 17× to 39×. It makes our fine-grain approaches very competitive to the state-of-the-art without any language, compiler or source access restrictions and the guarantee that the instrumented binary matches the original one.

6.3 Large-scale experiments

Methodology: To show FASE supports real workloads we perform a set of large-scale experiments. These tests consider the use of several DNN models, datasets and numerical datatypes. We report the validation accuracy after training, BLEU Score, or perplexity depending on the workload type. We compare the obtained accuracies against the reference implementation using FP32. We use FASE to emulate three different numerical formats in order to demonstrate the versatility of our tool:

- **BF16** with RNE rounding used until now.
- The mixed-precision (**MP**) [21,28] approach that employs FP32 precision in batch normalization and weight update layers. And performs FMA instructions using BF16 source inputs for the multiplication and an FP32 input for the accumulator, returning an FP32 value as output.
- A compound datatype that represents FP32 values using a tuple of BF16 values (**BF16x2**) [14]. Note that this format requires changing the original instruction with ad-hoc code that performs the operation using BF16x2.

We consider the following object classification models: ResNet18, ResNet34, ResNet50, ResNet101 [13], and MobilenetV2 [32] on CIFAR100 datasets [25].

Table 3: Large-scale experiments using FASE

Model	Dataset	Accuracy			
		FP32	BF16	MP	BF16x2
ResNet18	CIFAR100	71.91%	71.46%	71.89%	71.95%
ResNet34	CIFAR100	73.21%	72.83%	73.86%	72.66%
ResNet50	CIFAR100	74.78%	69.24%	74.25%	72.57%
ResNet101	CIFAR100	75.93%	67.10%	75.65%	76.00%
MobileNetV2	CIFAR100	75.04%	73.92%	75.16%	74.82%
AlexNet	ImageNet	60.79%	57.80%	60.18%	N/A
Inception	ImageNet	74.01%	72.03%	73.73%	N/A
LSTMx2 (Perplexity)	PTB	86.86	137.69	87.09	86.90
Transformers (BLEU)	IWSLT16	34.53	34.86	34.66	34.65

FASE attaches to Caffe framework to train AlexNet [24] and InceptionV2 [35] models, we use the same versions of tools as with the ResNet50 test on Section 6.2. Finally, we consider a full test on the same two natural language processing models as in Section 6.2. The whole set of hyper-parameters to train all of the models are detailed in the Supplementary Material.

Results: Table 3 shows the results of using FASE for several full DNN training workloads. We compare the accuracy of each network using our tool emulating different numerical formats (BF16, MP and BF16x2), and FP32.

With FASE we can determine if a reduced precision format is able to achieve the desired level of accuracy. When training object classification models on CIFAR100 with the BF16 numerical datatype, we observe significant drops in accuracy because the reduced number of mantissa bits in the BF16 numerical format fails to capture important information, especially on accumulations between distant numbers [15]. These drops are even higher on deeper models, for example, in ResNet101 there is an accuracy loss of 8.82% with respect to FP32. However, when FASE emulates MP using BF16 inputs and FP32 accumulators, these drops disappear, keeping the same levels of accuracy as FP32. The column BF16x2 shows results for a new compound datatype proposed by Henry et al. [14] that we emulate using FASE, it enables computing using BF16 arithmetic exclusively. In this case, we also observe good accuracy, on par with FP32.

Additionally, we emulate AlexNet and Inception training processes, FASE's results again show that using the BF16 numerical datatype is not enough to achieve comparable accuracy with respect to FP32. For AlexNet we measure an accuracy drop of 2.99%, while Inception model loses 1.98%. When emulating MP using FASE, we measure a boost on the accuracy reaching similar levels as FP32 for AlexNet and Inception, having drops of just 0.61% and 0.2% respectively.

Finally, FASE emulates the training of two natural language processing models. For the Transformer model we measure the BLEU score, higher is better. We observe that all the emulated numerical formats lead to accurate BLEU scores

when compared to FP32. Transformer-based models are known to display robust numerical properties and are resilient to numerical noise [36]; therefore, we can obtain state-of-the-art results using BF16. For the LSTMx2 model we measure *perplexity* (lower is better); the BF16 approach stops converging after 13 epochs giving NaN as result, we register the last perplexity value of 137.69, this confirms that LSTM models are not good candidates to use BF16 exclusively.

However, when we emulate approaches such as MP or BF16x2, we again obtain results comparable with FP32. These set of results on large-scale workloads illustrates the potential of FASE to emulate different numerical formats and to extract conclusions on their applicability. FASE can also be employed to study scenarios where numerical precision is changed at runtime depending on application progress, and to study other custom floating-point representations; making it a compelling fast, accurate and seamless tool.

7 Conclusions

The use of reduced precision numerical formats to lower computational costs and increase compute throughput has shown good results in the context of HPC workloads. More recently, the same principle is leading to a myriad of proposals for custom reduced precision numerical formats, both floating-point and integer, to improve the large computational and energy costs of training DNN.

Prior tools and methodologies to emulate reduced precision formats cannot deliver a fast, accurate and seamless experience when training DNN workloads. In this paper we propose FASE, an emulation tool for custom numerical formats. FASE is: (i) *accurate* by leveraging DBT techniques to emulate formats at instruction operand level; (ii) *fast* as it enables emulation of unmodified applications on large input sets thanks to a set of optimizations that lower its overheads significantly; and (iii) *seamless* as it works on any application or DNN framework without any language, compiler or source access restrictions and the guarantee that the instrumented binary matches the original one.

Our evaluation demonstrates that FASE is more accurate than other state-of-the-art proposals that employ coarse-grain emulation, uncovering relative errors that appear only in fine-grain emulation. We demonstrate that by applying both the *basic block* and *vectorization* optimizations, FASE latency overheads are manageable, ranging between 17× to 39× for a wide variety of workloads. These latencies enable the evaluation of large-scale unmodified workloads, which illustrate the potential of FASE to emulate different numerical formats and to extract conclusions on their applicability.

Acknowledgements 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. Adrià 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. Abadi, M., Agarwal, A., Barham, e.a.: TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems (2016)
2. ARM: SVE Instructions (sep 2021), https://bit.ly/3xC2B8E
3. Brown, T.B., Mann, B., Ryder, N., et al.: Language Models are Few-Shot Learners (2020)
4. Chatelain, Y., Petit, E., de Oliveira Castro, P., Lartigue, G., Defour, D.: Automatic Exploration of Reduced Floating-Point Representations in Iterative Methods. In: Euro-Par. LNCS, Springer (2019)
5. Chollet, F., et al.: Keras. https://github.com/fchollet/keras (2015)
6. Courbet, C.: Nsan: A Floating-Point Numerical Sanitizer. In: Proceedings of the 30th ACM SIGPLAN (2021)
7. Dawson, A., Düben, P.D.: rpe v5: An Emulator for Reduced Floating-Point Precision in Large Numerical Simulations. Geoscientific Model Development (2017)
8. Denis, C., de Oliveira Castro, P., Petit, E.: Verificarlo: Checking Floating Point Accuracy Through Monte Carlo Arithmetic. In: ARITH 2016
9. Durmus, A.U.: Replication of Recurrent Neural Network Regularization by Zaremba (Sep 2019), https://github.com/ahmetumutdurmus/zaremba/
10. Fu, Y., Guo, H., Li, M., et al.: {CPT}: Efficient DNN Training Via Cyclic Precision. In: ICLR (2021)
11. Fu, Y., You, H., Zhao, Y., et al.: Fractrain: Fractionally Squeezing Bit Savings Both Temporally and Spatially for Efficient DNN Training. In: NeurIPS (2020)
12. Gordic, A.: Original PyTorch Transformer Model (Oct 2020), https://github.com/gordicaleksa/pytorch-original-transformer
13. He, K., Zhang, X., Ren, S., Sun, J.: Deep Residual Learning for Image Recognition. CoRR (2015)
14. Henry, G., Tang, P.T.P., Heinecke, A.: Leveraging the BFLOAT16 Artificial Intelligence Datatype for Higher-Precision Computations (2019)
15. Higham, N.J.: The Accuracy of Floating Point Summation. SIAM (1993)
16. Huang, G., Liu, Z., van der Maaten, L., Weinberger, K.Q.: Densely Connected Convolutional Networks (2018)
17. Intel: Intel Caffe Framework Optimization, https://github.com/intel/caffe
18. Intel: Intel Deep Neural Network Library, https://github.com/intel/mkl-dnn
19. Intel: OneAPI DNN Library, https://oneapi-src.github.io/oneDNN/v1.4/index.html
20. Intel: Intel Math Kernel Library (2020), https://software.intel.com/en-us/mkl
21. Kalamkar, D., Mudigere, D., Mellempudi, e.a.: A Study of BFLOAT16 for Deep Learning Training (2019)
22. Khattak, G.R., Vallecorsa, S., Carminati, F., Khan, G.M.: Particle Detector Simulation Using Generative Adversarial Networks with Domain Related Constraints. In: ICMLA (2019)
23. Kiar, G., et al.: (December 2021), https://github.com/verificarlo/fuzzy
24. Krizhevsky, A., Sutskever, I., Hinton, G.E.: ImageNet Classification with Deep Convolutional Neural Networks. In: NeurIPS (2012)
25. Krizhevsky, A.: Learning Multiple Layers of Features from Tiny Images. (2009)

26. Loroch, D.M., Wehn, N., Pfreundt, F.J., Keuper, J.: Tensorquant - A Simulation Toolbox for Deep Neural Network Quantization (2017)
27. Luk, C.K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V.J., Hazelwood, K.: PIN: Building Customized Program Analysis Tools with Dynamic Instrumentation. ACM SIGPLAN Notices (2005)
28. Micikevicius, P., Narang, S., Alben, e.a.: Mixed Precision Training. ICLR (2018)
29. Molchanov, P., Tyree, S., Karras, T., Aila, T., Kautz, J.: Pruning Convolutional Neural Networks for Resource Efficient Inference (2017)
30. Paszke, A., Gross, S., Chintala, S., Chanan, G.: PyTorch (2020), https://github.com/pytorch/pytorch
31. Rodriguez, A., Ziv, B., Fomenko, E., Meiri, E., Shen, H.: Lower Numerical Precision Deep Learning Inference and Training (oct 2018), https://intel.ly/32G5WrT
32. Sandler, M., Howard, A., Zhu, M., Zhmoginov, A., Chen, L.C.: Mobilenetv2: Inverted Residuals and Linear Bottlenecks (2019)
33. Sun, X., Choi, J., Chen, C.Y., Wang, N., Venkataramani, S., Srinivasan, V., Cui, X., Zhang, W., Gopalakrishnan, K.: Hybrid 8-bit Floating Point (HFP8) Training and Inference for Deep Neural Networks. In: NeurIPS (2019)
34. Sun, X., Wang, N., Chen, C.Y., Ankur, J.M.N., Xiaodong, A., Swagath, C., Kaoutar, V., Maghraoui, E., Srinivasan, V., Gopalakrishnan, K.: Ultra-Low Precision 4-bit Training of Deep Neural Networks. In: NeurIPS (2020)
35. Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., A., R.: Going Deeper with Convolutions. In: CVPR (2015)
36. Tambe, T., Yang, E.Y., Wan, Z., Deng, Y., Reddi, V.J., Rush, A.M., Brooks, D., Wei, G.Y.: Adaptivfloat: A Floating-Point Based Data Type for Resilient Deep Learning Inference (2019)
37. Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, L., Polosukhin, I.: Attention is All You Need (2017)
38. Zaremba, W., Sutskever, I., Vinyals, O.: RNN Regularization (2015), arXiv
39. Zhang, T., Lin, Z., Yang, G., Sa, C.D.: QPyTorch: A Low-Precision Arithmetic Simulation Framework (2019)