Adapting Floating-Point Precision to Accelerate Deep Neural Network Training

John Osorio Ríos
john.osorio@bsc.es
Supervisors : Marc Casas, Adrià Armejach

Department of Computer Architecture

Universitat Politècnica de Catalunya
Barcelona Supercomputing Center (BSC)

This dissertation is submitted for the degree of

Doctor of Philosophy

July 2023

A ti papá, a ti mamá. A ti Helena ...

Agradecimientos

Con este documento se finaliza una larga etapa de mi vida. Si bien ha estado llena de altibajos, me quedo con todas y cada una de las enseñanzas que he obtenido. Este trabajo no se trata de un logro individual, significa todo lo contrario, tampoco diría que culmina mi vida académica sino más bien que le aporta a lo que para mí es un escalón más. Durante estos cinco años de trabajo, he estado rodeado de gente maravillosa que le ha brindado a mi vida en muchos aspectos, no sólo en lo académico sino en lo personal. He interactuado con personas de muchas otras culturas y he aprendido de sus costumbres, de tomar lo mejor de ellos y de brindarles un poco de mi experiencia. Quiero agradecer particularmente a mis advisors Marc y Adrià quienes me aportaron muchísimo conocimiento durante todo este tiempo, y nunca se cansaron de explicarme una y otra vez todo aquello que no entendía, y siempre estuvieron atentos a como mi proceso iba evolucionando. También debo mencionar a Eric, quien me transmitió muchos conocimientos que me permitieron avanzar muchísimo en lo académico, también como no, mencionar a Greg quien con una vasta experiencia pudo hacer recomendaciones neurálgicas durante mi investigación.

Desde lo familiar, a ellos, como no mencionarlos porque han estado no sólo durante estos años, sino durante toda mi vida. Prestos a ayudarme, a brindar una palabra de aliento para luchar por las cosas que a veces no iban tan bien. A ayudar a levantarme cuando veía las cosas de manera equivocada. A toda mi familia les agradezco mucho por la finalización de esta etapa académica.

Y como no mencionarla a ella, a una persona, que ha luchado a mi lado de una forma desinteresada, a quien en los momentos más difíciles de mi vida ha estado ahí, atenta a cada cosa que podría necesitar. Quien se decidió conmigo a iniciar una aventura bastante desconocida, pero que de seguro nos ha hecho mejores. Muchas gracias Helena, te agradezco con todas mis fuerzas por no haber permitido rendirme. Gracias por estar ahí ...

Acknowledgements

With this document, a long period of my life ends. Although it has been full of ups and downs, I remain with every one of the teachings that I have obtained. This work is not about individual achievement, it means the opposite, nor would I say that it culminates my academic life but instead that it contributes to what for me is one more step. During these five years of work, I have been surrounded by wonderful people who have given my life in many aspects, not only academically but also personally. I have learned to interact with people from many other cultures and to learn from their customs, to take the best of them, and give them a bit of my experience. I want to particularly thank my advisors Marc and Adrià who have provided me with a lot of knowledge during all this time, they never get tired of explaining to me over and over again everything that I did not understand, and they were always attentive to how my process was evolving. I must also mention Eric, who gave me a lot of knowledge that allowed me to advance a lot academically, as well as not to mention Greg who, with vast experience, was able to make neuralgic recommendations during my research.

I need to mention to my family, they have been there not only during these years but throughout my life. Ready to help me, to offer a word of encouragement to fight for things that sometimes did not go well. To help me get up when I saw something the wrong way. I am very grateful to all my family for completing this academic stage.

And how not to mention her, a person who has selflessly fought by my side, who has been there in the hardest moments of my life, attentive to everything I might need. Who decided with me to start this unknown adventure, but that surely has made us better. Thank you very much, Helena, I thank you with all my strength for not allowing me to give up. Thanks for being there ...

Table of contents

List of figures

List of tables

Glossary

Roman Symbols

FASE	Fast, Accurate and Seamless Emulator
$\mathbf{FMA^{bf16}_{n\_m}}$	BFloat16 operator with n BF16 input adder and m BF16 input multiplier
$\mathbf{FMA^{bf16}_{1\_1}}$	BFloat16 operator with 1 BF16 input adder and 1 BF16 input multiplier
$\mathbf{FMA^{bf16}_{1\_3}}$	BFloat16 operator with 1 BF16 input adder and 3 BF16 input multiplier
$\mathbf{FMA^{bf16}_{1\_2}}$	BFloat16 operator with 1 BF16 input adder and 2 BF16 input multiplier
$\mathbf{FMA^{bf16}_{3\_3}}$	BFloat16 operator with 3 BF16 input adder and 3 BF16 input multiplier
$\mathbf{FMA^{bf16}_{2\_2}}$	BFloat16 operator with 2 BF16 input adder and 2 BF16 input multiplier
AI	Artificial Intelligence
AMX	Advance Matrix eXtensions
BF16	Brain Float 16 bits
BN	Batch Normalization
BSC	Barcelona Supercomputing Center
CERN	European Council for Nuclear Research
CNN	Convolutional Neural Network
CPU	Central Processing Unit
DBT	Dynamic Binary Tool
DNN	Deep Neural Network

ECML	European Conference on Machine Learning
EMA	Exponential Moving Average
FLOP	Floating Point Operation
FMA	Fused Multiply Addition
FPGA	Field Programmable Gate Array
FP	Floating Point
GAN	Generative Adversarial Network
GB	Giga Byte
GCC	GNU C Compiler
GeV	Giga Electron Volts
GNN	Graph Neural Network
GPT	Generative Pretrained Transformer
GPU	Graphics Processing Unit
HEP	High Energy Physics
HPC	High Performance Computing
ICMLA	International Conference on Machine Learning and Applications
IEEE	The Institute of Electrical and Electronics Engineers
ISA	Instruction Set Architecture
ISPASS	International Symposium on Performance Analysis of Systems and Software
LHC	Large Hadron Collider
LSTM	Long Short-Term Memory
MKL	Math Kernel Library
ML	Machine Learning
MNIST	Modified National Institute of Standards and Technology database

MP	Mixed Precision
NDCG	Normalized Discounted Cumulative Gain
NGCF	Neural Graph Collaborative Filtering
NLP	Natural Language Processing
NMT	Neural Machine Translation
OS	Operative System
PTB	Penn Tree Bank
RNE	Round to Nearest Even
RNN	Recurrent Neural Network
SGD	Stochastic Gradient Descent
SR	Stochastic Rounding
TETC	IEEE Transactions on Emerging Topics in Computing
TFLOP	Tera Floating Point Operation
TPU	Tensor Processor Unit
VAE	Variational Autoencoder
WGAN	Wasserstein Generative Adversarial Network
WLCG	Worldwide Large Hadron Collider Computing Grid
WU	Weight Update

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 for understanding the properties of new numerical formats on a particular workload. However, current state-of-the-art techniques cannot perform these tasks quickly and accurately on a wide variety of workloads.

The usage of Mixed Precision (MP) arithmetic with *floating-point* 32-bit (FP32) and *16-bit half-precision* aims at improving memory and *floating-point* operations throughput, allowing faster training of bigger models. This is one of the most used techniques, and has been successfully applied to train DNNs. Despite its advantages in terms of reducing the need for key resources like memory bandwidth or register file size, it has a limited capacity for diminishing further computing costs, as it requires 32-bits to represent its output. On the other hand, full half-precision arithmetic fails to deliver state-of-the-art training accuracy.

Several hardware companies are proposing native Brain Float 16-bit (BF16) support for neural network training. Fused Multiply-Add (FMA) functional units constitute a fundamental hardware component to train DNNs. Its silicon area grows quadratically with the mantissa bit count of the computer number format, which has motivated the adoption of the 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 FP32 format to achieve state-of-the-art training accuracy, which limits the impact of adopting BF16.

To address all of the previous concerns with respect to different numerical formats, specific training techniques, and how to increase the use of reduced precision approaches, this Thesis proposes FASE, a Fast, Accurate, and Seamless Emulator that leverages dynamic binary translation to enable emulation of custom 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.

To show the advantages of FASE we test it in object classification, natural language processing, and generative networks workloads. We use FASE to analyze BF16 usage in the training phase of a 3D Generative Adversarial Network (3DGAN) simulating High Energy Physics detectors. FASE allows us to confirm that BF16 can provide results with similar accuracy as the *full-precision* 3DGAN version and the costly reference numerical simulation using double-precision arithmetic.

We use FASE to characterize and analyze computer arithmetic to propose a seamless approach to dynamically adapt floating point arithmetic. Our dynamically adaptive methodology enables the use of full half-precision arithmetic for up to 96.4% of the computations when training state-of-the-art neural networks; while delivering comparable accuracy to 32-bit floating point arithmetic. Microarchitectural simulations indicate that our *Dynamic* approach accelerates training deep convolutional and recurrent networks with respect to FP32 by 1.39$\times$ and 1.26$\times$, respectively.

Finally, we propose an approach able to train complex DNNs entirely using the BF16 format. Using FASE we introduce 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$\times$ 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.

In summary, this work evaluates and proposes several reduced numerical approaches ranging from BF16, MP, Dynamic, and compound data types to train different DNN networks. We present FASE as a binary analysis tool to emulate custom numerical formats avoiding costly hardware implementations while getting accurate emulation of large workloads without requiring any source code modification.

Chapter 1

Introduction

Since AlexNet [55] was used to solve the ImageNet challenge the interest in Convolutional Neural Networks (CNN) to manage image classification problems increased considerably. Several CNNs had been used to solve this challenge; some examples of these models are ZFNet [103], GoogLeNet/Inception [88], VGGNet [82] and Residual Networks (ResNet) [34].

In the same way, an increasing interest in other problems like object detection, image segmentation, machine translation, or speech recognition has been studied in recent years using Deep Neural Networks (DNN) to solve them.

However, employing DNNs is computationally intensive, especially in the quantity of Fused Multiply Add (FMA) instructions used during the training and inference processes. For this reason, during the last years there is an increasing interest in using reduced precision numerical data representation formats to train DNNs [48, 86, 87, 99].

The idea behind all of these research approaches is to reduce the power consumption, the number of operations, or the execution time, without losing accuracy. However, many methods use complex hardware, software, or both to accomplish their goals. Ultimately, the original DNN model must be changed entirely to keep accuracy, making its implementation difficult for scientists who know little about numerical data representations.

Recent research about mixed precision training [51, 64] shows that these techniques can do training and inference processes using 16-bit floating-point operations without accuracy loss while keeping the original model almost unchanged. There are other approaches where even lower precision schemes are used to check if a training process is possible without hurting accuracy. Some implementations use 4 or 8 bits in specific training layers to represent floating point numbers [25, 26].

An increasing number of custom numerical formats are becoming available to study and to see how their properties could be used to train DNN models. However, their implementation in real hardware is restrictive, and the cost becomes high if we implement them all to check their advantages and disadvantages. Here is where an emulation approach can alleviate the effort required to understand the numerical properties of these numerical approaches without having to rely on expensive, in time and cost, hardware implementations.

We propose a binary analysis tool called FASE (Fast, Accurate, and Seamless Emulator) based on Intel PIN, which helps to emulate custom numerical formats and to test their numerical properties during the training phase. FASE successfully emulates data types like Brain Float 16 (BF16), Mixed Precision (MP), dynamic techniques, and a new set of custom FMA operators. With FASE we can train several convolutional, generative, and sequential models using Caffe, Tensorflow, and PyTorch.

1.1 Contributions and Thesis Organization

While completing this thesis, we developed a Fast, Accurate, and Seamless Emulation (FASE) approach to emulate numerical datatypes without incurring complex changes in the source code of frameworks for DNN training. FASE is designed to work seamlessly with models implemented in multiple frameworks, such as Caffe, PyTorch, and TensorFlow. The contributions can be summarized as follows:

- A numerical datatype emulator based on a binary analysis tool.
- A mixed numerical precision approach to train generative adversarial networks.
- A dynamic precision datatype scheme to train DNNs.
- A new set of BF16 operators that rely exclusively on 16-bit arithmetic operations to train DNNs.

The contributions are divided into two topics; the first is an emulator with several use cases. The second one uses this emulator to study different numerical precision schemes to train multiple state-of-the-art DNN models.

Below, we show the thesis organization and also complemented with the list of contributions covering Chapters 4 to 7:

- **Chapter 1 - Introduction.** Motivates our work from a general point of view. It mentions the advantages of emulation to test numerical datatypes and gives some context about the work we develop.
- **Chapter 2 - Background and Related Work.** Introduces some important topics about each of the main chapters in this thesis. The idea is to give some context to the reader.

- **Chapter 3 - Experimental Setup.** Details the hardware and software we use to do our tests. Additionally, it covers the methodology followed to emulate each numerical data type.
- **Chapter 4 - FASE: A Fast, Accurate and Seamless Emulator for Custom Numerical Datatypes.** Explains how we implement our custom numerical emulator and presents how it performs with respect to native executions.
- **Chapter 5 - Evaluating Mixed-Precision Arithmetic for 3D Generative Adversarial Networks to Simulate High Energy Physics Detectors.** It is the first use case to present the advantages of FASE while emulating Mixed-Precision (MP) to simulate a High Energy Physics Detector at CERN.
- **Chapter 6 - Dynamically Adapting Floating-Point Precision to Accelerate DNN Training.** We develop a dynamic precision approach that changes between a higher precision numerical scheme to a lower one following the evolution of the training loss.
- **Chapter 7 - A BF16 FMA is All You Need for DNN Training.** We present the first full BF16 numerical approach to train several DNNs without using FP32. We emulate a new family of FMA operators using FASE.
- **Chapter 8 - Conclusions and Future Work.** Concluding remarks about the work done, we discuss future work to keep FASE updated with new families of floating point precision schemes.

1.2 List of Publications

1.2.1 Conference Papers

- **Evaluating Mixed-Precision Arithmetic for 3D Generative Adversarial Networks to Simulate High Energy Physics Detectors.**
  John Osorio, Adrià Armejach, Gulrukh Khattak, Eric Petit, Sofia Vallecorsa and Marc Casas.
  In 19th IEEE International Conference on Machine Learning and Applications (ICMLA), Dec 2020.
- **Dynamically Adapting Floating-Point Precision to Accelerate Deep Neural Network Training.**
  John Osorio, Adrià Armejach, Eric Petit, Greg Henry and Marc Casas.

In 20th IEEE International Conference on Machine Learning and Applications (ICMLA), Dec 2021.

- **FASE: A Fast, Accurate and Seamless Emulator for Custom Numerical Formats.**
John Osorio, Adrià Armejach, Eric Petit, Greg Henry and Marc Casas.
In European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases (ECMLPKDD), Sep 2022.

1.2.2 Journal Papers

- **A BF16 FMA is All You Need for DNN Training.**
John Osorio, Adrià Armejach, Eric Petit, Greg Henry and Marc Casas.
In IEEE Transactions on Emerging Topics in Computing (TETC). Jul-Sep 2022

1.2.3 Posters

- **FASE: A Fast, Accurate and Seamless Emulator for Custom Numerical Formats.**
John Osorio, Adrià Armejach, Eric Petit, Greg Henry and Marc Casas.
In IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), May 2022.

Chapter 2

Background and Related Work

This section contains the background that helps to cover each of the main chapters in the Thesis. It helps to understand what exists in the state-of-the-art and the main concerns when using reduced precision approaches.

2.1 Importance of DNN Workloads

Following the progress made in image classification tasks [18, 58] by Khrizhevsky et al. [55], Szegedy et al. [88] and He et al. [34], DNNs [57] have been successfully applied to domains like biology [49], economy [3], chemistry [62], or sports management [47]. Many refined techniques have emerged to tackle these new problems beyond image classification. Among them, the Generative Adversarial Networks (GAN) proposed by Goodfellow et al. [29]. CERN has recently proposed to drive High Energy Physics (HEP) simulations using GANs [13, 52, 53, 75, 80].

DNNs are becoming ubiquitous in different areas, as mentioned before. CNNs can accurately detect and classify objects over large image data sets [55], and Recurrent Neural Networks (RNNs) using encoder-decoder models are capable of solving tasks like Neural Machine Translation (NMT) [6]. However, to achieve the desired accuracy levels, many samples must be exposed to the model tens or hundreds of times during training. This fact increases training costs regarding power, memory storage, or compute time.

Due to DNN models increasing popularity and current trends on DNNs the training costs will continue to grow as state-of-the-art DNNs feature increasingly large parameter counts [8, 9, 70]. Training process improvements became necessary to reduce costs. The fused Multiply-Add (FMA) functional unit is a key hardware component to train DNNs since they support most floating-point instructions required for DNN training [90].

2.2 Machine Learning in Scientific Applications

2.2.1 High Energy Physics

The High Energy Physics (HEP) community has a long tradition of using Neural Networks and Machine Learning methods (Random Forests, BDT, MLPs) to solve specific tasks, in particular, related to a more efficient selection of exciting events over the overwhelming background produced at colliders such as the Large Hadron Collider (LHC). In recent years, several studies have demonstrated the benefit of using Deep Learning (DL) to solve typical tasks related to data taking and analysis. Building on these examples, many HEP experiments are now working on integrating DL into their workflows for many different applications: from data quality assurance [5] to real-time selection of interesting collision events [27], to simulation [28] and data analysis [78]. For example, generative models, from GAN to VAE, are being tested as fast alternatives to Monte Carlo-based simulation. Anomaly detection algorithms are being explored to improve data quality monitoring, to design searches for rare new-physics processes, or to analyze and prevent faults in complicated systems such as detectors and accelerator control systems.

Training of such models has been made tractable with the improvement of optimization methods and the advent of dedicated hardware well adapted to tackle the highly-parallelizable task of training neural networks. However, memory consumption still represents a limiting factor in many applications: HEP detector output is represented by millions of read-out channels and is usually too large to be processed. The standard approach in these cases is segmenting and selecting regions of interest for further processing. Analyzing larger input samples and processing deeper models represents an advantage in data processing efficiency, accuracy, and exploitation of computing resources.

2.2.2 Generative Adversarial Networks and HEP

A Generative Adversarial Network (GAN) is a type of generative modeling that uses deep learning methods, like CNN. The idea behind these GAN models is to generate new content based on datasets used to train them [29]. A GAN comprises two networks competing against each other during the training phase. One of the networks is called a **Generator** and the other **Discriminator**. The Generator is trained to generate new data that becomes almost real. The Discriminator is trained to discover which data from the Generator is real. When the discriminator has a 50% of the probability of making a mistake, we could say that the model is correctly trained. Once the GAN is trained, we could use the Generator at inference to Generate new data.

Variational Autoencoders (VAE) are also generative models that can help generate new data based on a training dataset. The idea behind VAE, similar to GAN, is to have two networks. One of them is called the **Encoder** and is followed by a second one that is named the **Decoder** [21]. The encoder takes the inputs to the VAE and creates an intermediate representation with less information than the input but is efficient enough to contain its main properties. Then, the decoder will try to recover the original input with this information. With the encoder output, also known as latent space, we could take random points from it and then feed the decoder to generate new data.

HEP simulations with detailed Monte Carlo methods are highly resource-intensive, consuming more than 50% of the Worldwide LHC grid (WLCG) resources [11]. In the past few years, developing and optimizing generative models to speed up or entirely replace Monte Carlo simulation has become a lively research field. CALOGAN [75] simulated particle showers in a simplified calorimeter conditioned on the primary particle energy ($1-100$ GeV). DL was used for fast simulation of the ATLAS calorimeter [80]. Showers with energies $1-260$ GeV and pseudorapidity $|\eta|$ in the range of $0.2-0.25$ were generated using both VAE and GAN. The showers were generated as a flattened array of pixels. The GAN-generated showers were reported to have a better agreement with the Monte Carlo data than the VAE-generated showers. DijetGAN [20] used GAN to simulate diject events, a background process for critical physics studies at LHC. WGAN simulated the LHC detector output collapsed into a two-dimensional array of cells [13]. A pre-trained regressor network was incorporated into the GAN setup. The results were reported to be promising, although more work was required for practical implementation.

3DGAN [52] is one of the most realistic and accurate models for simulating a HEP detector. The showers are generated for particles within the $2-500$ GeV energy range, with several orders of magnitude speedup. Due to the generation of 3D images, large training data, and a deeper model, the training can benefit significantly from reducing the data resolution. On the other hand, scientific simulation is also a performance-critical task and must retain high accuracy.

2.3 Mixed Precision Training

Exploring lower-precision numerical formats for DNN training has been an active research topic in recent years. Some works propose non-standard data formats as an alternative to FP32. For example, prior work indicates that dynamic *fixed-point* is effective in training DNN with *low-precision* multipliers [15]. This approach obtains *state-of-the-art* results by uniformly applying the dynamic *fixed-point* format with different scaling factors, which are driven by the overflow rate displayed by the *fixed-point* numbers. Gupta et al. [97] show the benefits of applying stochastic rounding to 16-bit *fixed-point* multiply and add operators. This work relies on FPGA emulation to show the benefits of stochastic rounding when applied to a custom fully connected neural network dealing with the MNIST dataset. Wang et al. [96] propose to train DNNs using 8-bit *floating-point* (FP8) numbers by relying on a combination of 8-bit and 16-bit arithmetic and by using stochastic rounding to obtain *state-of-the-art* results. Additionally, Sun et al. [84] present a novel approach similar to the previous one, but without stochastic rounding; however, they require a quantization and two newly defined FP8 numerical data types.

Recently released and announced hardware products support new numerical data types to train DNNs. For instance, Nvidia GPUs support Mixed Precision (MP) training by leveraging their tensor cores, which combine FP16 and FP32 in their Volta architecture [64] and more recently proposed the TF32 format using up to 19-bits [67]. Similarly, Kalamkar et al. [51] propose an MP scheme using BF16 and FP32 formats. They emulate this approach using library calls that must be integrated into the target DNN models. The conversion from BF16 to single precision does not require a scaling factor, as both types cover the same range. Conversion from 32 to 16 bits requires an RNE rounding operation.

FMA instructions implementing an MP approach bring significant benefits since they require less memory bandwidth and register storage than FP32 FMAs. Additionally, when using BF16, there are no substantial modifications or new hyperparameters for the targeted DNN model. It operates on the same value ranges as FP32, allowing trivial adoption of this approach by the community.

2.4 Dynamic Precision Training

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

For example, prior work indicates that dynamic fixed-point is sufficient to train DNNs with low precision multipliers [15]. This approach obtains state-of-the-art results by uniformly applying the dynamic fixed-point format with different scaling factors driven by the overflow rate displayed by the fixed-point numbers. Applying SR to 16-bit fixed-point multiply and add operators has been beneficial [31]. Other proposals train DNNs using 8-bit floating point (FP8) numbers by relying on a combination of 8-bit and 16-bit arithmetic and using SR to obtain state-of-the-art results [96] or propose a multi-precision approach also using SR to keep DNN training accuracy [77]. Finally, there is a newer approach that does not require SR [84]; however, a quantization step and two newly defined FP8 numerical data types are needed.

Other proposals focus on leveraging available hardware support to achieve the same objectives. This is the case of the MP proposals that employ half-precision representation such as FP16 and BF16 for tensors [51, 64]. The BF16 numerical format has been used in specific-purpose hardware targeting DNNs [97] and will soon be supported by off-the-shelf hardware from Intel and Arm. Our dynamic training approach falls under this category. As we demonstrate, it can be applied on top of FP32 or MP training to reduce memory bandwidth and computational requirements while delivering comparable accuracy. Furthermore, FASE and the *Dynamic* approach we proposed can be adapted and used to explore and validate other existing and future encodings.

2.5 Reduced Precision Floating-Point Formats

As the complexity of new DNN models increases [9], training these models translates into large computational and environmental costs [25], valued at millions of dollars. Multiple proposals try to reduce these costs by using reduced precision strategies to take advantage of the favorable area and power trade-offs associated with narrower hardware units [91].

Several studies show that reduced precision techniques [85, 91] diminish computing time and energy consumption when training DNNs. Micikevicius et al. [64] proposes an MP technique using FP16 on Nvidia GPUs, while Kalamkar et al. [51] use BF16; however, both require FP32 arithmetic to compute critical sections such as WU and BN layers. Additionally, Graphcore [92] presents a hardware accelerator that targets MP training with FP16 datatypes. They also define a new numerical datatype called AI-Float, which uses stochastic-rounding hardware to maintain accuracy across models. While these methods achieve comparable accuracy concerning 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. [85] propose a complex methodology for ultra-low precision training using a 4-bits numerical format. This proposal delivers worse precision than the state-of-the-art. It 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 for 32-bit accumulations. Moreover, this approach requires an ad-hoc recipe to train each network, severely undermining its generality and applicability.

Fu et al. [25, 26] 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) computing 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 [101, 105].

In contrast, our $\text{FMA}^{\text{bf16}}_{\text{n\_m}}$ proposal (see section 7) relies on the already well-known and widely adopted BF16 datatype to offer different precision tiers. Additionally, applying $\text{FMA}^{\text{bf16}}_{\text{n\_m}}$ does not require additional ad-hoc steps that vary depending on the DNN model. $\text{FMA}^{\text{bf16}}_{\text{n\_m}}$ can be deployed in low-end computing systems as it requires a small and cheap BF16 FMA unit. $\text{FMA}^{\text{bf16}}_{\text{n\_m}}$ provides different accuracy levels, and $\text{FMA}^{\text{bf16}}_{\text{1\_2}}$ or $\text{FMA}^{\text{bf16}}_{\text{2\_2}}$ arithmetic can be used when $\text{FMA}^{\text{bf16}}_{\text{1\_1}}$ is insufficient, thereby lowering training costs concerning conventional FP32 arithmetic while delivering state-of-the-art accuracy.

Chapter 3

Experimental Setup

In this chapter, we talk about the methodology we follow. We also cover the standard hardware setup followed during the thesis and the software used to implement each test to get the results we analyze in each chapter.

3.1 Methodology

We decided to follow an incremental methodological approach. To adapt different floating-point methods, we created a standard tool called FASE (Fast, Accurate, and Seamless Emulator); check Chapter 4. FASE uses Intel PIN [44] to analyze each of the instructions to be computed in the processor during the Deep Neural Network process.

We did several tests, training different state-of-the-art DNNs. This chapter explains the main models with the hyperparameters we used throughout the thesis development. In the following chapters, we describe any additional studied workload if needed.

3.2 Experimental Setup

To train and test the CNN implementations that use ImageNet [18] dataset, we use the Intel-Caffe [38] framework (version 1.1.6a). We use the Intel MKLDNN [39] (version 0.18.0) Deep Neural Network library and the Intel MKL library [43] (version 2019.0.3) to run numerical kernels since both libraries are optimized to run on our testing infrastructure. We use the *py-Caffe* python interface to define and run the experiments, which loads the data and orchestrates the execution.

Finally, to perform the Natural Language Processing (NLP) models and CIFAR [54] datasets experiments, we use PyTorch [76] (version 1.4.0), Intel MKLDNN (version 0.21.1) and the

Intel MKL library (version 2019.4). We use torchtext to manage the pre-processing steps needed in the *seq2seq* model.

3.2.1 Neural Network Models

Object Classification Models

To evaluate our proposals, we consider three representative state-of-the-art CNN models: AlexNet [55], Inception V2 [45, 88] and ResNet-50 [34]. These models are the backbone of recently published highly successful models [100].

We use the ImageNet database [18] as training input. To keep execution times manageable when using FASE, we run the experiments using a reduced ImageNet Database, similar to the Tiny ImageNet Visual Recognition challenge data set [24]. Therefore, we use 256,000 images divided into 200 categories for training and 10,000 images for validation. The images have no modifications in terms of size. All the evaluated CNN models remain unmodified.

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

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

ResNet50 is a network that delivers good accuracy and avoids the vanishing gradients issue using residual blocks. We train it using a multi-step approach. The batch size is 64, and the base learning rate is 0.05, updated every 30 epochs. The gamma hyperparameter, momentum value, and weight decay are set to 0.1, 0.9, and 0.0001, respectively. The training process runs for a total of 32 epochs.

Additionally, we consider ResNet18, ResNet34, ResNet50, ResNet101 [34], and MobilenetV2 [81] on both CIFAR10 and CIFAR100 datasets [54]. We use the hyperparameter setup recommended by the state-of-the-art [85]. We train all networks for 160 epochs using the SGD Optimizer. We use a batch size of 128 and an initial learning rate of 0.1, dividing by 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.

Natural Language Processing Models

To test the performance of our approach over RNNs, we selected a *seq2seq* model to solve the NMT task [6]. This model is trained using the Multi30K dataset [23]. This dataset has 30,000 multilingual English-German sentences. We take 29,000 sentences as the training set and 1,000 for validation purposes. The model has 20,000,000 parameters in total; we use a batch size of 256 and the Adam Optimizer to do the training process. The model was built using Gated Recurrent Units (GRU). Training is done over ten epochs.

The LSTMx2 model [102] applied to the PTB dataset is also used. We follow the details in [102] to train the medium-size model; our test uses the source code available in [22]. 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, and a sequence length of 35, and a dropout equal to 0.5.

A transformer-based model [94] was 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 $\varepsilon = 10^{-9}$. We use a batch size of 12000 and 4000 warm-up steps. We use the source code available in [30]. All additional details are in [30, 94].

Finally, we train a simple transformer-based model on the Multi30k dataset [23] to translate between English and Dutch. This implementation uses the source code available in [7]. We train this model for ten epochs using the Adam optimizer with a fixed learning rate of $5 \cdot 10^{-4}$ and a batch size of 128.

3DGAN Model

A custom test was done using 3DGAN [53] with Tensorflow [1] 1.15 and Keras 2 [14]. We use the same MKL and MKLDNN libraries as in the ResNet50 case. The 3DGAN network is trained for 60 epochs 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 [53].

Recommender System

We consider a recommender system based on Graph Neural Networks (GNN), the Neural Graph Collaborative Filtering (NGCF) [98]. This model is trained using the MovieLens ML-100k dataset [33]. We train it for 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 [63].

3.3 Hardware Setup

We used Marenostrum4 hosted at Barcelona Supercomputing Center, a supercomputer based on Intel Xeon Platinum processors from the Skylake generation. It is a Lenovo system composed of SD530 Compute Racks, an Intel Omni-Path high-performance network interconnect, and running SuSE Linux Enterprise Server as the operating system. Its current Linpack Rmax Performance is 6.2272 Petaflops.

This general-purpose block consists of 48 racks housing 3456 nodes with 165,888 processor cores and 390 Terabytes of main memory. Our experiments use one full node of this system. Compute nodes are equipped with the following:

- 2 sockets Intel Xeon Platinum 8160 CPU with 24 cores each @ 2.10GHz for 48 cores per node.
- L1d 32K; L1i cache 32K; L2 cache 1024K; L3 cache 33792K
- 96 GB of main memory 1.880 GB/core, 12x 8GB 2667Mhz DIMM (216 nodes high memory, 10368 cores with 7.928 GB/core).
- 200 GB local SSD available as temporary storage during jobs

Chapter 4

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

4.1 Introduction

As mentioned in Section 2.1 state-of-the-art DNNs are using huge training parameter counts. There are already approaches to reducing the training computation costs via mechanisms that incur accuracy degradations [37, 65, 81]. Additionally, there are approaches able to reduce training costs without reducing DNNs accuracy. These approaches rely on reduced computer number formats [25, 26, 84, 85]. 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 [59], 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 [104] to address the PyTorch framework. The intrinsic approach rounds each individual floating point operation and displays a latency of 50$\times$ with respect to native executions. It is a source-level approach that can be used to evaluate all implementations of neural networks based on Tensorflow. All of these approaches are designed to target specific DNN frameworks and require changes to the framework and model source code. Other tools like Verificarlo [12] 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 [60] 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 [38], Tensorflow [1], and PyTorch [76]. 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 behavior before committing to costly hardware implementations. This chapter makes the following contributions:

- We propose FASE[1], 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.

4.2 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 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. These workloads can tolerate well low-precision formats in certain computations, with proposals that go as low as 4-bit numerical representations [25, 26, 85].

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

Table 4.1 Comparison of state-of-the-art proposals.

Features	Emulators				
	RPE [17]	QPyTorch [104]	TensorQuant [59]	Verificarlo [12]	FASE
Fast	✗	✓✓	✓	✓✓	✓
Accurate	✓	✗	✓	✓	✓
Seamless	✗	✗	✗	✗(recompilation)	✓
Dynamic Libraries	✗	✗	✗	✗(Lib. recompilation)	✓
Independent	✗	✗	✓	✗(compiler dep.)	✓

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, 64, 79]; 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 4.1 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.

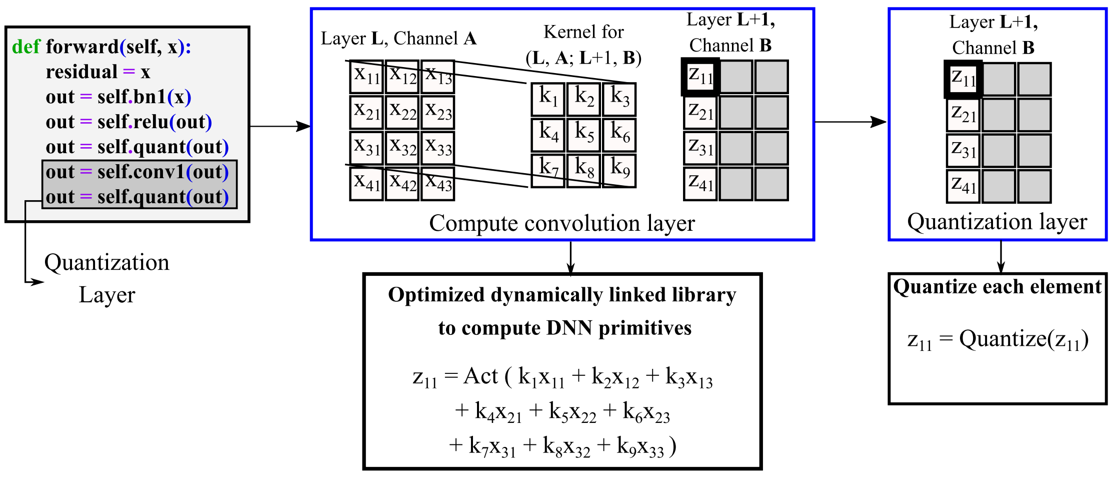
The Reduced Precision Emulator (`rpe`) [17] is an emulator which supports reduced precision that can be computed on the available hardware format and rounding. The tool operates in a fine-grain manner. They report overheads from 10-70× on small emulated workloads. However, like all other source level approaches, code modification interfere with compiler optimizations impacting numerical accuracy [19]. Furthermore, it is currently restricted to Fortran applications. `Verificarlo-Vprec` [12, 19] 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 [12]. 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 [59] and QPyTorch [104]. 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 [25, 26] use QPyTorch [104] 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 4.3, the implementation and performance optimization in Section 4.4, the strategy we apply to evaluate the tool on machine learning frameworks in Section 4.5 and evaluate accuracy and performance in Section 4.6.

4.3 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



def forward(self, x):
residual = x
out = self.bn1(x)
out = self.relu(out)
out = self.quant(out)
out = self.conv1(out)
out = self.quant(out)
Quantization Layer
Layer L, Channel A
Kernel for (L, A; L+1, B)
Layer L+1, Channel B
Compute convolution layer
Layer L+1, Channel B
Quantization layer
Optimized dynamically linked library to compute DNN primitives
$z_{11} = \mathrm{Act}(k_1x_{11} + k_2x_{12} + k_3x_{13} + k_4x_{21} + k_5x_{22} + k_6x_{23} + k_7x_{31} + k_8x_{32} + k_9x_{33})$
Quantize each element
$z_{11} = \mathrm{Quantize}(z_{11})$

Fig. 4.1 Steps for coarse-grain emulation on a convolutional layer.

which contain key optimized routines.

Figure 4.1 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 [40]. 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 as 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.

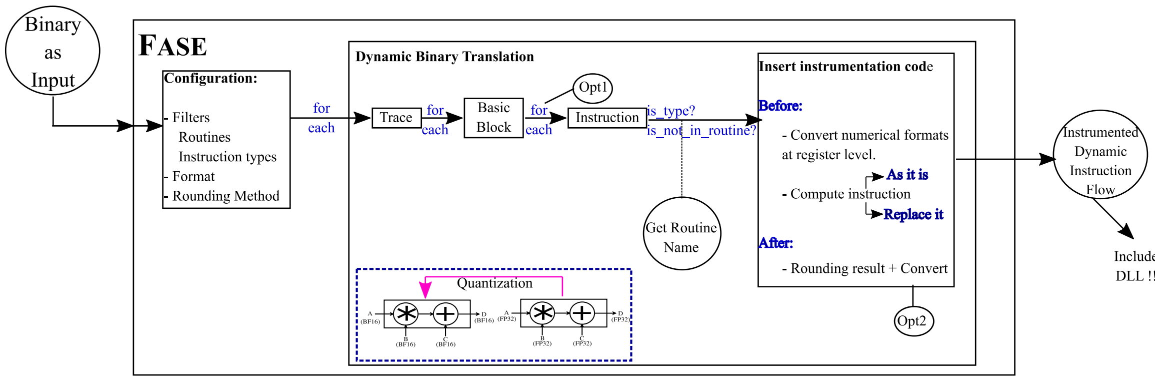


Fig. 4.2 FASE Implementation Overview

4.4 FASE Implementation

4.4.1 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.

Figure 4.2 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.

Listing 4.1 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
```

Once the code has been instrumented at the basic block level, the next step is *analysis*. During the analysis step the instrumented dynamic instruction flow, which includes any external libraries, is executed. The analysis is the most computationally expensive step as the modified instruction flow with code insertions is executed.

4.4.2 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.

1. **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, 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.

2. **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.

3. **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.

Listing 4.2 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;
}
```

4.4.3 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 4.6. On the other end, the fully optimized version will be referred as `full opt`.

1. **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 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 4.1 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 4.2 and Section 4.6.2 we refer to this optimization as `Opt1`.

2. **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 4.2 presents the vectorized optimization FASE implements to boost the performance, reducing the emulation latency as Section 4.6.2 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 4.2 and 4.6.2 section we refer to this optimization as `Opt2`.

4.5 Applying FASE to DNN Training Workloads

The main use case for FASE in this chapter 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 [51], while others target existing hardware [64].

4.5.1 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 [35]. 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 [16].

4.5.2 DNN training strategies

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

1. **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 [51, 64] training technique, which is similar to using BF16 but uses the FP32 representation to do the accumulation step on FMA instructions.
2. **Using routine filters:** Certain functions (or DNN layers) require higher accuracy than others. For this reason, FASE enables applying different numerical format conversions or avoiding 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 behavior via simple configuration options.
3. **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.

4.6 FASE Evaluation

Our experimental methodology considers the evaluation of FASE on several DNN frameworks as explained in Chapter 3.

4.6.1 Emulation accuracy

Methodology

We use a single precision matrix multiply (SGEMM) present in DNN training processes to evaluate emulation accuracy. We implement this benchmark that multiplies two matrices using the Intel Math Kernel Library (oneMKL) [43]. We compiled the source code using GCC 8.1 with all the AVX512 optimizations active on our platform. 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 output matrix element 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 [104] and TensorQuant [59]. Secondly, we attach FASE to the benchmark binary, which instruments the code from the dynamically linked Intel MKL library. This enables the execution of the workload using FASE fine-grain emulation at the 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.

Results

The following results illustrate that the fine-grain approach is much more accurate in emulating reduced precision numerical formats. Figure 4.3 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

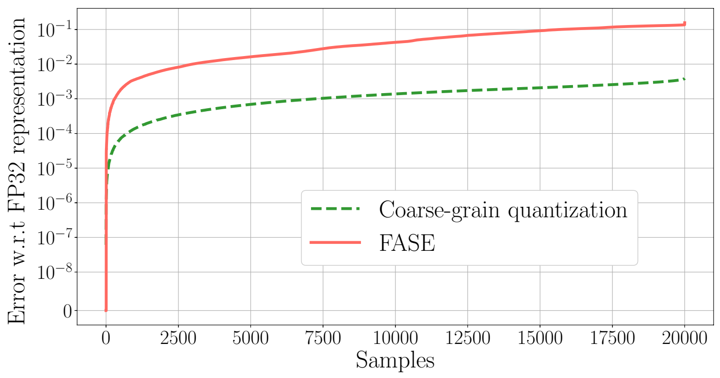


Fig. 4.3 Relative error of fine-grain and coarse-grain emulation methodologies (for BF16 and RNE rounding) with respect to native FP32 execution.

coarse-grain method cannot capture the errors that accumulate per instruction; however, FASE can track these errors and deliver a result that is much closer to reality.

In Section 4.6.3, 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.

4.6.2 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 4.4.3. We compare each version for all benchmarks against a reference native FP32 execution without instrumentation. Additionally, we report FASE's instrumentation overhead, which 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 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 described in Section 4.6.1. Then we evaluate FASE on the following machine learning workloads:

- ResNet50 for one batch of size 64, on ImageNet, as explained in Chapter 3
- CERN 3DGAN [53] for one batch of size 128, as explained in Section 3.2.1

Table 4.2 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×

Finally, we consider two Natural Language Processing models:

- LSTMx2 model [102] on the PTB dataset. We train one batch of the medium-sized model with a batch size of 20, check Section 3.2.1
- A transformer-based model [94] applied to the IWSLT16 dataset, details are in Section 3.2.1

Results

Table 4.2 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 emulating 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 measures 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.

4.6.3 Large-Scale Experiments

Methodology

We perform large-scale experiments to show that FASE supports real workloads. These tests consider using 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 to demonstrate the versatility of our tool:

- **BF16** with RNE rounding used until now.
- The mixed-precision (**MP**) [51, 64] 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 [35] that represents FP32 values using a tuple of BF16 values (**BF16x2**). Note that this format requires changing the original instruction with ad-hoc code that uses BF16x2.

We consider the following object classification models: ResNet18, ResNet34, ResNet50, ResNet101 [34], and MobilenetV2 [81] on CIFAR100 datasets [54]. FASE attaches to the Caffe framework to train AlexNet [55], InceptionV2 [88], and ResNet50 as explained in Chapter 3. Finally, we consider a full test on the same two natural language processing models as in Section 4.6.2. The whole set of hyper-parameters to train all models is detailed in Chapter 3.

Table 4.3 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

Results

Table 4.3 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 [36]. 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. [35] 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, and BAT'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 the 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 [89]; 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. This 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.

4.7 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 section, we propose FASE, an emulation tool for custom numerical formats [74]. 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.

Chapter 5

Evaluating Mixed-Precision Arithmetic for 3D Generative Adversarial Networks to Simulate High Energy Physics Detectors

5.1 Introduction

Models based on DNNs must be trained on large sets of data to reach acceptable accuracy levels. Training incurs huge computational and memory training costs. Several novel techniques are aimed at decreasing the training costs by avoiding the use of FP32 numerical datatype and replacing it with non-standard low precision data representation formats [15, 31, 96].

Recent hardware products support *16-bit half-precision* to accelerate training. Micikevicius et al. [64] and Kalamkar et al. [51], show how to train DNNs using MP approaches. A combination of FP32 and *half-precision* representations using either the *floating-point* 16 (FP16) or the BF16 datatypes, which we display in Figure 5.1, reaches the same accuracy levels as FP32 arithmetic for many DNN applications. A strong emphasis is given to *Fused Multiply-Add* (FMA) instructions computing $A \cdot B + C$, representing the most significant part of the computational time. The MP approach for FMAs uses FP32 precision to accumulate the contributions of $A \cdot B$ and C, while the multiplication uses *half-precision* inputs, as seen in Figure 5.2.

In this chapter, we adapt and demonstrate the capacity of MP arithmetic to train GANs driving HEP simulations. In particular, we focus on MP FMAs combining BF16 and FP32, which Figure 5.2b represents.

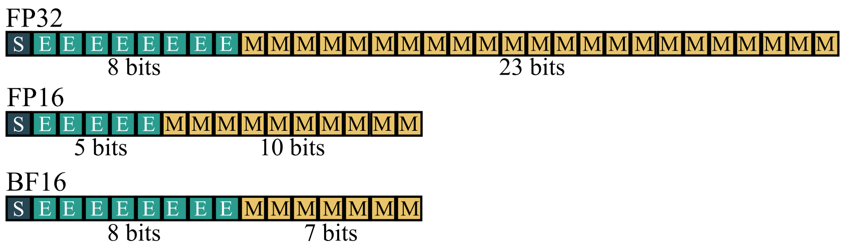
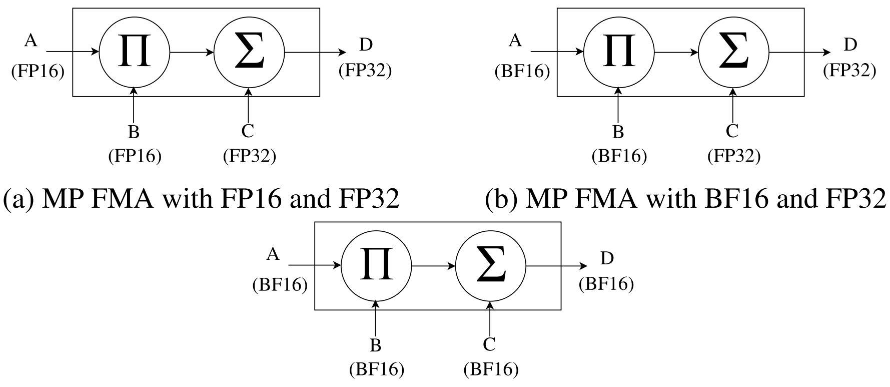

FP32
S E E E E E E E E M M M M M M M M M M M M M M M M M M M M M M M
8 bits
23 bits
FP16
S E E E E E M M M M M M M M M M
5 bits
10 bits
BF16
S E E E E E E E E M M M M M M M
8 bits
7 bits

Fig. 5.1 *Floating-point* Formats. `S` sign bit, `E` exponent bits, `M` mantissa bits.



(a) MP FMA with FP16 and FP32 (b) MP FMA with BF16 and FP32

(c) BF16 FMA

Fig. 5.2 Different alternatives when computing a *Fused Multiply-Add* (FMA) instruction.

We use FASE to emulate lower precision numerical formats without the need for native hardware support. FASE intercepts FMA instructions and modifies its operands at the register level to emulate *low-precision* arithmetic. In particular, we use the RNE to convert FP32 values to the BF16 format. Once the conversion is done, FASE tool gives the control back to the training process, which implies that the FMA instruction is natively executed with some of its operands rounded to 16-bit representation. FASE can be applied to either the PyTorch [76], the Caffe [46], or the Tensorflow [1] frameworks.

Our evaluation shows that 98% of *floating-point* instructions are FMAs and that we just need to compute in full FP32 the 0.001% of them, as Section 5.4 indicates. In addition, we empirically validate that MP FMAs using BF16 are able to train GAN models with similar accuracy as full FP32, and a real physical Monte Carlo reference simulation. Section 5.5 contains this evaluation. To the best of our knowledge, this is the first proposal that trains GAN models using MP and applies them to HEP simulation problems.

5.2 Mixed Precision Training

BF16 is a numerical datatype created by Google used in their Google Brain group [97]. It was initially created to reduce the amount of data shared by different computing nodes during distributed DNN training, and used specifically on their Tensor Processing Units (TPU). However, since BF16 has the same dynamic range as FP32, as both have the same number of exponent bits (see Figure 5.1), the conversion between these two formats is very simple and it shows great robustness while training DNNs. On the other hand, the *floating-point* 16 (FP16) data type requires scaling factors to perform the conversion from/to FP32. For these reasons, BF16 has become a reference datatype with hardware support announced for multiple upcoming computer architectures from vendors like Nvidia [67], Intel [42], and Arm [83].

However, training with BF16 requires careful attention to arithmetic sensitive parts of the algorithm such as weights updates, batch normalization, and gradients summation. This leads to the usage of a MP approach, where a combination of FP32 and BF16 is used in different places and flavors. When employing MP, FMA instructions (MP FMA) combine *half-precision* and FP32 input values and usually accumulate in FP32 as shown in Figure 5.2.

Figure 5.2a details the type of FMA instruction employed in Nvidia's MP training [64]. An FMA instruction computes $D = A \cdot B + C$. Input parameters A and B are represented in FP16, the result is added to the third input C, typically a weight of the master copy represented in FP32. The final output D is also represented in FP32. This approach requires additional steps to enforce that FP32 values that are converted into FP16 fall into the representative range of the latter by applying a scaling factor.

Figure 5.2b shows the FMA implementation used by Kalamkar et al. in [51], which is similar to the one previously described. The only difference is about the *low precision* datatype used to do the calculations which is BF16, simplifying the conversion from/to FP32. Since the BF16 data type is only available on the proprietary Google TPU, this study of MP employing BF16 was performed without hardware support, by modifying the evaluated neural networks to perform library calls to do the conversion from FP32 to BF16 in software, using truncation and RNE rounding. In this regard, Intel's next generation family of Intel Xeon processors, code-named Cooper Lake, will incorporate BF16 hardware support to use MP approaches in the training processes of DNN [42].

Additionally, there is another possible FMA instruction implementation, as shown in Figure 5.2c, where all inputs and outputs are represented using BF16. However, using this approach during training does not provide good converge properties for the most popular DNNs. This is caused by the absorption issue during the add step in the FMA instruction. That is, when *right-shifting* the smaller of the two operands to align bits with the same exponents, the lowest bit cannot be represented in the rounded result. The smaller number can even be entirely absorbed due to the limited number of *mantissa* bits, this issue is known as *swamping* [96] or *absorption* in the computer arithmetic field, and leads to information loss when the magnitudes of the added operands have a difference $> 2^{mantissa+1}$ bits. In the case of BF16 with 7 *mantissa* bits, the *swamping* threshold is 256.

Regardless of the MP FMA implementation used, some phases of the training process require FP32 precision to keep the same accuracy when training the DNN models. This is the case of the weight updates (WU), batch normalization (BN), and soft-max layers.

We demonstrate the feasibility to apply MP training to GAN models for HEP simulations. The following section introduces a three-dimensional GAN model and outlines the computational challenges it presents, which can be mitigated by applying MP. Therefore, the use of MP can help advance this field by enabling larger input datasets or models with additional complexity that can be trained in a sensitive amount of time, with the objective to achieve better simulation results.

5.3 MP Emulation Tool

Without available hardware implementing the BF16 numerical format, several approaches have been used to emulate the behavior of reduced *floating-point* representations. Most notably, these efforts have focused on libraries that perform transformations like truncation and rounding [12, 51]. However, such an approach requires a significant effort to port every new target neural network to use these libraries, leading to a tedious process that needs to be repeated for each network. To solve these issues, we use FASE.

FASE performs the following steps:

- Checks the current execution routine to determine if we are executing routines that belong to weight update or batch normalization layers. In that case, computation proceeds in FP32.
- The tool intercepts all *floating-point* instructions of the workload, including FMAs. For each FMA instruction, operands that need to be rounded to BF16, depending on the current routine, are rounded using the RNE algorithm.
- The tool can dynamically change its operation mode via a simple inter-process communication method that can be invoked from the python high-level interface; this is useful to test some additional scenarios or to avoid instrumenting all the preprocessing steps involved during the training process of DNN.

Figure 4.2 shows an overview of the steps done by FASE. FASE seamlessly converts the FMA inputs A and B to BF16 using RNE rounding on all forward pass computations. As

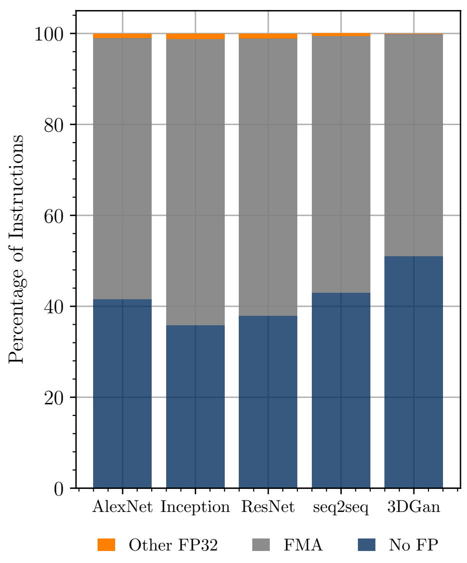


Fig. 5.3 Instruction Breakdown for multiple neural network models.

defined in MP training, these are then accumulated in FP32 (see Figure 5.2b). Similarly, in the backward step, the FMA inputs A and B are again converted to BF16, and the final result in FP32 contains the weight gradients. Finally, the weight gradients will be used to update the weights using FP32 arithmetic, which is crucial [51, 64, 96] to obtain the same accuracy as *state-of-the-art* FP32 training methods. These steps are repeated for each batch during the training process of the 3DGAN model. Note that the amount of computation (FMAs) done during the forward and backward passes is orders of magnitude larger than the computation done in the layers that employ FP32 arithmetic. More details about this are in Section 5.4.1.

5.4 Evaluation Methodology

To train and test the 3DGAN implementation, we consider the information in Section 3.2.1.

5.4.1 Use case characterization

Figure 5.3 shows the percentage of instructions processed by several DNN models. While training DNNs, the compute units (GPU, CPU) perform substantial additions and multiplications due to the nature of neural network architectures. Ultimately, these are calculated using specialized hardware instructions inside the computing units called FMAs. As shown in Figure 5.3, FMA instructions represent a large portion of the overall instruction mix. The *Other FP32* category are *floating-point* instructions that are not FMA, and *No FP* are integer,

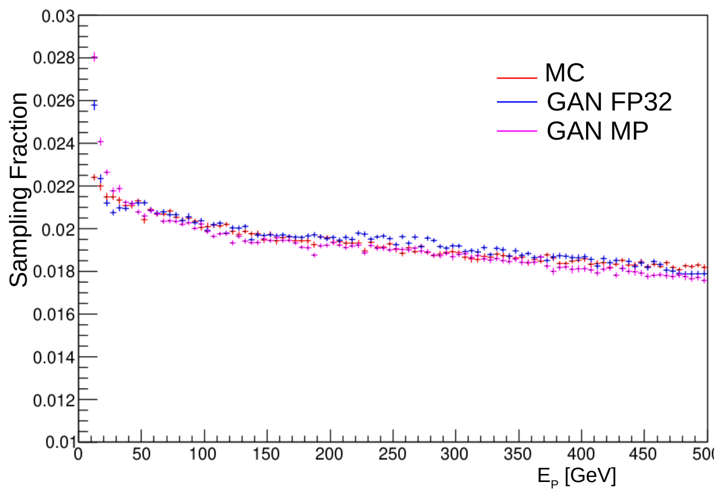


Fig. 5.4 Sampling Fraction (SF) vs. particle energy profile histograms for simulation using Monte Carlo (MC) (red), GAN FP32 (blue), and GAN MP (magenta).

memory, and branch instructions.

In the figure, well-known DNN models such as AlexNet [55], Inception [88], ResNet [34] and seq2seq [6, 93] are shown to stress the importance FMA instructions have on any DNN training process. The percentage of FMA instructions for these models is 57.42%, 60.93%, 62.95%, and 56.44%, respectively. In 3DGAN, our use case, $374 * 10^9$ instructions are executed during one training batch, the percentage of FMAs is 48.80%, which is a significant portion, opening an opportunity to apply a technique such as MP to improve throughput and reduce bandwidth requirements. Note that, as mentioned before, certain layers or routines still need to be computed using FP32 (weight updates, batch normalization, and soft-max); however, the number of FMAs attributed to these layers or routines is not significant, less than 0.001% of the total number of FMAs for the 3DGAN model.

5.5 Results

The 3DGAN aspires to generate scientific data leveraging an approach borrowed from the domain of visual images. It is therefore highly crucial to understand if these techniques can retain the high level of accuracy required for a scientific simulation. Each simulated image represents the pattern of energy depositions in a HEP detector when a primary particle enters the detector volume. This pattern of energy depositions also known as “shower” has distinct physical features. These features are important information pertaining to a certain type of particle, the energy with which it strikes the detector, as well as the geometry of the detector.

The 3DGAN performance is validated by comparing GAN images to the Monte Carlo data

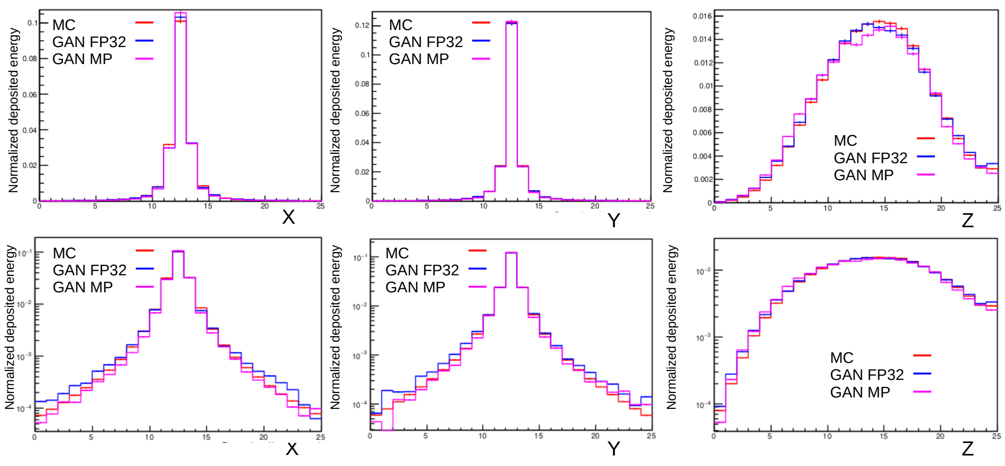


Fig. 5.5 Shower shapes (energy distribution) for Monte Carlo (red), GAN FP32 (blue), and GAN MP (magenta) along X, Y, and Z axis. In linear scale(top row); and in log scale (bottom row).

in terms of several relevant physics distributions. Here we further compare features from images generated by networks trained using FP32 and the MP data types. The Monte Carlo data set used in this experiment represents the geometry of a sampling calorimeter. In a sampling calorimeter there are alternate arrays of passive and active layers. Energy is only recorded in the active layers and thus is a fraction of the energy of the incident particle. The sampling fraction (SF) is the ratio of the energy recorded in the detector to the energy of the incoming particle (E_p): it represents the detector response. Figure 5.4 presents the sampling fraction calculated from images simulated using Monte Carlo and 3DGAN. There is a high level of agreement for most of the E_p range for the GAN generated images using both the FP32 and MP approaches.

The geometry of the generated shower is often used in particle identification. There can be a number of ways to quantify geometrical features. The shower shapes are the distribution of the deposited energy along a certain detector axis. The Z axis is along the depth of the calorimeter, while X and Y axes are the transverse directions. Figure 5.5 compares the shower shapes for Monte Carlo images with images generated by the models trained using the FP32 and MP approaches. The top row shows the shower shapes on linear scale. The log scale plots in the bottom row are added to better appreciate the agreement at the tails of the showers. The training with the MP approach is able to achieve a similar level of accuracy as the higher precision 32-bit approach.

As explained in Chapter 4, we created FASE to emulate MP training, since the real hardware to perform the tests was not available at the time of the experiment. By using FASE, we have been able to demonstrate that MP training can be used for HEP simulations. MP delivers the same level of accuracy as higher precision approaches implemented using FP32. Furthermore, using real hardware implementations, this approach would reduce memory storage and bandwidth requirements by nearly 50% compared to full FP32 implementations, and increase the FMA throughput by at least twice for the same hardware vector register size. Performance improvements can be even better with dedicated wider units targeting BF16, such as the one available on the recently released Intel Cooper Lake, Nvidia Tensor Core or Google TPU.

5.6 Conclusions

The use of DNN to lower the computational requirements of Monte Carlo based HEP simulations has received a lot of attention in recent years. In particular, GAN models stand out as the most accurate solutions to simulate HEP detectors. However, as models in the HEP field become more complex, requirements for memory bandwidth and capacity increase. Moreover, in a production-ready GAN algorithm capable of generating data for full-scale detectors these memory-related challenges would be exacerbated.

The use of lower-precision numerical formats can help alleviate these constraints. However, HEP simulations require a high level of accuracy in order to minimize uncertainty in the final physics measurements. In this chapter, we undertake a study to determine if reduced precision training based on MP attains similar levels of accuracy as the default FP32 reference training.

To accomplish this, we propose FASE that enables the emulation of lower precision numerical formats without the need for hardware support. Using this tool we show that FMA instructions are responsible for a significant chunk of the total computational workload when training well-known DNN models, as well as our 3DGAN use case, for which FMAs account for 48.80% of the total instruction count. By training the 3DGAN network for 60 epochs using a representative dataset, we have been able to show that MP training employing the BF16 numerical format is able to deliver the same level of accuracy as higher-precision approaches implemented using FP32 [71].

Chapter 6

Dynamically Adapting Floating-Point Precision to Accelerate DNN Training

6.1 Introduction

As Section 2.1 stands, DNNs are widely used across different tasks, while their training costs are exponentially growing. Several proposals successfully mitigate training costs by replacing the use of standard FP32 arithmetic with alternative approaches that employ non-standard low precision data representation formats [15, 31, 96]; reducing memory storage, bandwidth requirements, and compute costs. Hardware vendors have incorporated half-precision data formats [51, 64] like the BF16 format [51] and have implemented MP instructions, which aim at reducing memory bandwidth and storage consumption.

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

We use FASE to characterize and analyze computer arithmetic usage in machine learning frameworks (e.g., PyTorch, Caffe, and Tensorflow) and emulate different floating point formats. We analyze multiple state-of-the-art CNN and RNN models and find that over 98,15% of the total floating point instructions are FMAs. To better analyze network behavior from a computer arithmetic perspective, we observe the network's requirements regarding floating

Table 6.1 state-of-the-art FMAs for training.

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

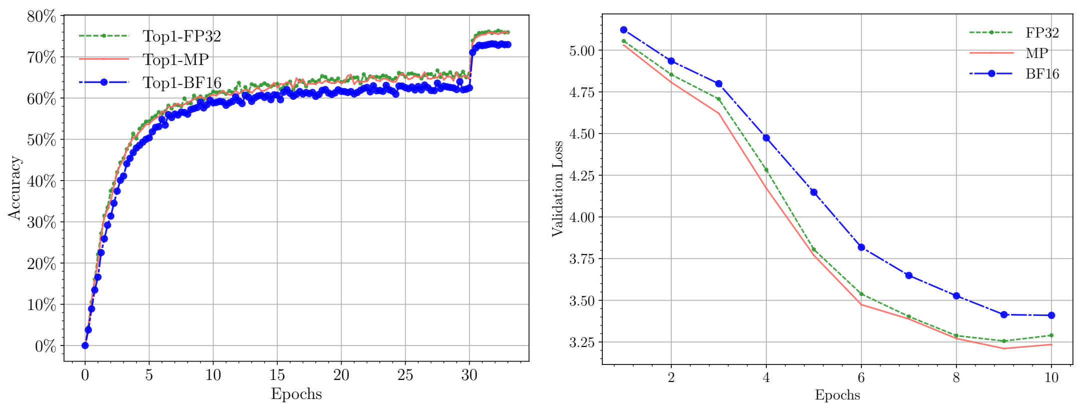
point precision in various phases of the training algorithm and learning epochs.

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

Our evaluation with FASE shows that the *Dynamic* approach can obtain comparable accuracy w.r.t FP32 and MP training for CNN and RNN models over the same number of epochs. For all evaluated CNNs, over 94.6% of the FMAs are performed entirely in half-precision (BF16), demonstrating that it is possible to use half-precision computations for a large portion of the training process without incurring any accuracy degradation. In addition, we show that our heuristics are sensitive to the network requirements, increasing the amount of MP (or FP32) computations when half-precision is not enough in terms of accuracy. Finally, we use the Sniper [10] architectural simulator to evaluate the performance benefits of our approach since there is no real hardware supporting full BF16 FMA instructions. Our evaluation shows that the *Dynamic* approach accelerates training by 1.39$\times$ and 1.26$\times$ over FP32 for CNN and RNN, respectively, while keeping the same accuracy levels.

6.2 Background in Mixed-Precision Training and Motivation

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



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

Fig. 6.1 Analysis for evaluated DNNs

Nvidia GPUs support MP training by leveraging their tensor cores, which combine the floating point 16-bit (FP16) and FP32 [64] formats in FMA instructions. Figure 5.2a displays an MP FMA instruction, which computes $D = A \cdot B + C$. Input parameters A and B are represented in FP16 and the result is added to the third input C, typically a 32-bit weight. The final output D is also represented in FP32. This approach requires applying a scaling factor when converting FP32 values to FP16 to avoid range representation issues.

Similarly, hardware vendors [51, 66] propose combining the BF16 and FP32 formats in a single FMA instruction, which Figure 5.2b shows. Conversion from FP32 to BF16 does not require a scaling factor as both types have the same representation range and, therefore, the conversion just requires applying Round to Nearest Even (RNE) rounding.

MP FMA instructions bring significant benefits since they require less memory bandwidth and register storage than FP32 FMAs. However, using a full BF16 FMA like the one represented in Figure 5.2c provides even larger benefits. In terms of memory bandwidth, BF16 FMAs require moving 50% and 33% less data than FP32 and MP FMAs, respectively. Similarly, BF16 FMAs require one-half and two-thirds of FP32 and MP FMAs register storage, respectively.

Table 6.1 summarizes the input/output data types and the precision employed during the arithmetic operations in three state-of-the-art MP approaches. Both Nvidia GPUs (tensor cores) and Google TPUs [50] take multiplication inputs in the half-precision format while the accumulator and accumulation arithmetic use FP32, producing an FP32 output. Recent Intel Xeon CPUs implement the new AVX512-BF16 extensions [41], which also feature half-precision multiply inputs but convert them to FP32 before the arithmetic operations, i.e., the FMA is entirely performed using FP32. Finally, we propose to use full BF16 FMA operations to train DNNs, which can provide significant savings in register storage, bandwidth, functional unit logic, and performance improvements by better leveraging data-level parallelism via vectorization.

To date, the implementation of full BF16 FMAs has not been adopted by hardware vendors, as training DNNs using BF16 FMAs does not provide state-of-the-art convergence properties, leading to lower accuracy with respect to MP and FP32 training. In this section, we describe a *Dynamic* approach that enables using BF16 FMAs for almost the entire training process while keeping the same convergence properties as FP32 and MP training. We apply all state-of-the-art FMA approaches to train four relevant models in Section 6.3, and we analyze the reasons why a full 16-bit approach does not provide enough accuracy. Based on these observations, we describe our *Dynamic* precision approach in Section 6.4.

6.3 Analysis of State-of-the-Art Approaches

We analyze the different state-of-the-art approaches to perform FMA instructions on three CNN models: AlexNet [55], Inception V2 [45, 88], and ResNet-50 [34]; and an RNN model [6, 93] that solves the NMT task, referred in this work as sequence to sequence (*seq2seq*) model. Section 7.6 contains the details regarding the configuration of each evaluated model and the methodology we follow.

6.3.1 Instruction Counts

Figure 5.3 shows the instruction mix for one batch on each network. We observe that floating point instructions constitute a large portion of these workloads. For example, they represent 58.44%, 60.93%, 62.95% and 56.44% of the total count for AlexNet, Inception V2, ResNet-50, and *seq2seq*, respectively. The amount of non-FMA FP instructions remains below 1.10% for the four networks. Therefore, FMA instructions constitute a large portion of the whole training workload. These measurements justify the focus on FMA instructions.

Prior research [51, 64] describes the need to use FP32 arithmetic in WU routines and BN layers when using training approaches based on MP arithmetic. We run an experimental campaign to confirm their observations and to measure the number of instructions devoted to WU and BN. In the case of ResNet-50, this instruction count is around 30 million instructions per batch, that is, just 0.04% of the total FP instructions. AlexNet and Inception V2 present similar results. In the case of the *seq2seq* model, this instruction count remains low, 1.60% of the total FP instructions. In conclusion, this data motivates our efforts to reduce the cost of FMA instructions for DNN training, since WU calculations and BN layers, which must be computed using FP32 arithmetic, represent a negligible portion of the training workload.

6.3.2 Static Approaches for Training

State-of-the-art training methods [51] employ the same data representation format for a certain variable throughout the whole training process, hence the term static. Similarly, we also evaluate a static approach that performs all FMA instructions in full BF16 except for WU calculations and BN layers, which are performed in FP32. Figure 6.1a shows the three different static training techniques on ResNet-50. The methodology we use to generate Figure 6.1a is described in Section 6.6. We observe that MP achieves very similar accuracy as FP32. In contrast, the BF16 approach does not deliver the desired level of accuracy with a significant drop, around 3%, w.r.t FP32 and MP. Figure 6.1b shows the same three techniques on the *seq2seq* model. Further details on the model appear in Section 6.6. Again, FP32 and MP present similar validation loss curves, while the BF16 approach fails to deliver comparable results.

An FMA entirely relying on BF16 precision can potentially provide large performance improvements. However, as Figures 6.1a and 6.1b illustrate, full BF16 training fails to deliver competitive levels of accuracy.

6.3.3 FMAs Data Representation Requirements

We analyze FMA data representation requirements using ResNet-50 training in order to explain the accuracy drop of full BF16 in Figures 6.1a and 6.1b.

Since the addition step of FMA instructions requires right-shifting the smaller of the two operands by the difference of the exponents, it is possible to completely eliminate the smallest number if the exponent difference is larger than the amount of *mantissa* bits. In the case of BF16, there are 7 *mantissa* bits. This issue is called *swamping* [96].

Figure 6.2 shows the exponent differences for all FMA instructions involved in ResNet-50 training on different epochs. It represents the percentage of FMAs (*y-axis*) that avoid entirely losing one of the two operands in the addition for a determined number of *mantissa* bits (*x-axis*). During the first epoch, just around 60% of the FMA instructions would not entirely lose one of the two addition operands when using the BF16 data representation. For epochs 8, 16, and 32, this percentage is slightly lower. In contrast, 23 mantissa bits, i.e., the standard FP32 representation, allows to keep the two addition operands of near 100% of the FMAs.

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

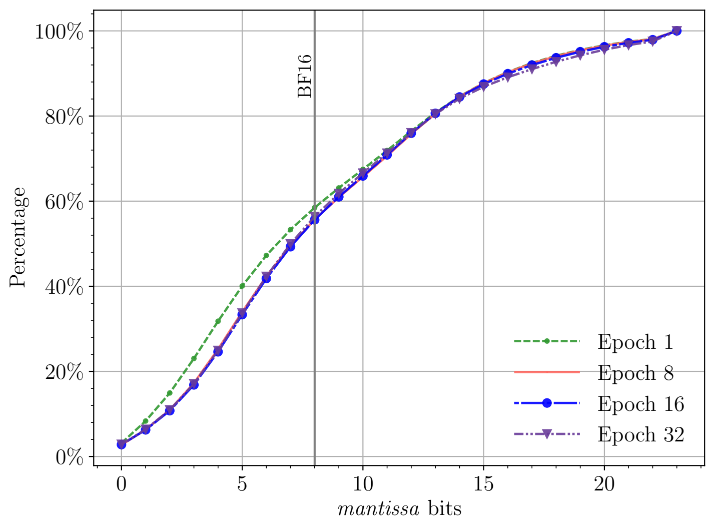


Fig. 6.2 Percentage of FMAs without *swamping*.

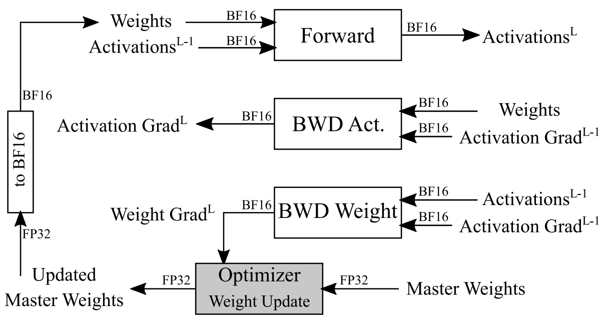


Fig. 6.3 BF16 training process for a layer.

6.4 Dynamic Precision Training

We propose to dynamically switch floating point precision during training to obtain the same accuracy as FP32 or MP while reducing training costs by employing full BF16 FMAs for as many batches as possible. Our strategy for DNN training uses BF16 FMAs as long as the training loss improves. When we detect training loss stagnation, we switch to a higher precision approach such as MP or FP32 for a number of batches, until training loss improves again. The use of reduced precision helps with the generalization capabilities of the model [31]. To detect training loss stagnation we calculate its Exponential Moving Average (EMA) [56] for a moving window of batches. We compare this EMA value with a threshold parameter to determine if training is progressing with the currently employed FMA format. The EMA is updated throughout the training process, guiding the decisions to switch between full BF16 and MP precision.

Algorithm 1 shows the high-level pseudo-code of our *Dynamic* precision training strat-

Algorithm 1 Dynamic Precision Training

```
1: $numBatchesMP \leftarrow 10$                                  // Number of consecutive MP batches
2: $numBatchesBF16 \leftarrow 1000$                              // Number of consecutive BF16 batches
3: $emaThreshold \leftarrow 0.04$                                // Defines EMA reduction threshold
4:
5: $precisionModeBF16 \leftarrow False$                          // Indicates current precision mode, True means BF16
6: $countBatchesBF16 \leftarrow 0$                               // Counts how many numBatchesBF16 have been executed
7: $numBatchesTrain \leftarrow numBatchesMP$                     // Number of batches per training loop iteration
8:
9: for $i = 0$ to $niter$ do
10:     train.step(numBatchesTrain)                              // numBatchesTrain batches precisionModeBF16
11:     $trainingLoss[i] \leftarrow train.trainingLoss$
12:     if $i = 5$ then                                          // Initial history to calculate EMA
13:         $EMA \leftarrow average(trainingLoss)$
14:     if $i > 5$ then
15:         $EMAprev \leftarrow EMA$
16:         $EMA \leftarrow emaCalculation(trainingLoss, EMAprev)$   // Each numBatchesMP
17:         if $(precisionModeBF16 != True)$ then
18:             if $((EMAprev - EMA) > emaThreshold)$ then       // If training loss goes down
19:                 $precisionModeBF16 \leftarrow True$
20:                 changeToBF16()                               // Switch precision to BF16
21:         else
22:             $countBatchesBF16 \leftarrow countBatchesBF16 + numBatchesTrain$
23:             if $(countBatchesBF16 = numBatchesBF16)$ then
24:                 if $((EMAprev - EMA) > emaThreshold)$ then   // If training loss goes down
25:                     $countBatchesBF16 \leftarrow 0$          // Stay in BF16 precision
26:                 else                                         // If training loss stagnates
27:                     $precisionModeBF16 \leftarrow False$
28:                     changeToMP()                             // Switch precision to MP
29:                     $countBatchesBF16 \leftarrow 0$
```

egy. The algorithm starts the training process using the state-of-the-art MP training [51] for several batches, defined by *numBatchesMP* parameter. Then, it computes the EMA of the training loss and, if its reduction is above a certain threshold (*emaThreshold* parameter), it computes the next *numBatchesBF*16 using BF16 training. Once training has processed these *numBatchesBF*16 batches, our algorithm updates the EMA value and compares it against the *emaThreshold* parameter again. If the EMA reduction is not large enough, the algorithm switches back to MP training. Otherwise, it keeps using BF16 training for *numBatchesBF*16 batches before updating the EMA value and comparing against it once again.

This method is applied during the entire training and it is able to dynamically adapt the way each batch handles its FMA instructions depending on the training loss progress. If such progress stagnates, the FMAs are computed using a more expensive higher-precision method like the MP or FP32 techniques. If training progresses well, the algorithm dynamically switches the way FMAs are computed and chooses the full BF16 approach. Figure 6.3 illustrates the implications of processing one batch using BF16 FMAs. Similar to MP: (i) weights, activations, and gradients are stored in BF16 format; (ii) an FP32 master copy of the weights is maintained and updated with the weight gradients during the optimizer step (e.g., in the Stochastic Gradient Descent solver); and (iii) reduction operations present in BN layers use FP32 arithmetic. In contrast, all arithmetic operations within the forward and backward passes are performed using full BF16 arithmetic. Section 6.7 demonstrates that dynamically switching the precision of batch execution between full BF16 and MP FMAs provides the same training convergence as FP32 or MP FMAs during the whole training achieving performance gains.

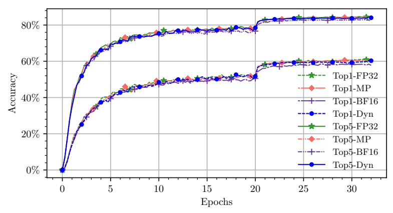
6.5 FASE: Fast, Accurate and Seamless Emulator

Due to the lack of available hardware implementing full BF16 FMAs, software emulation is required to study their numerical behavior. Several approaches have been used to emulate BF16 arithmetic, most notably via highly-tuned low-level libraries that truncate floating point operands [12, 51]. However, such approaches require access to the mathematical library upon which the training relies. And this should be done for all potentially different implementation frameworks such as Tensorflow, PyTorch, or Caffe. With the aforementioned approaches, these arithmetic modifications must be done on the source code or at compile time. Besides the effort that code modifications or recompilation of complex frameworks require, this methodology is not applicable to closed-source mathematical libraries, like Intel MKL. Since computer arithmetic is highly sensitive to the implementation, not instrumenting the true executed binary sequence of instruction can lead to wrong interpretation [19].

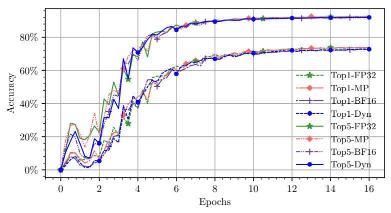
To overcome these limitations we used FASE. FASE performs the following steps: First, it checks the current FMA operation mode, which for the purposes of this chapter can be FP32, MP, or BF16 (see Figure 5.2). Additional numerical formats can be easily emulated. Second, it determines whether we are executing routines that belong to WU calculations or BN layers. If yes, computation proceeds in FP32. Third, the tool intercepts all floating point instructions of the workload, including FMAs. For each FMA instruction, FASE rounds off all operands that need to be converted to BF16 using an accurate round to nearest even algorithm. Finally, FASE can dynamically change its FMA operation mode via a simple inter-process communication method that can be invoked from the Python high-level DNN framework code were Algorithm 1 is implemented.

6.6 Experimental Methodology

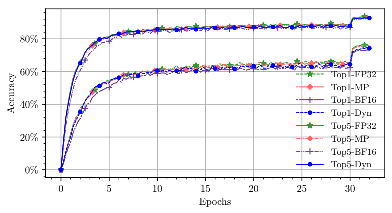
Chapter 3 contains detailed information regarding the software and hardware used to train the different DNN models used in this chapter with the whole set of hyperparameters. Additionally, we explain here a specific approach we take into account to test this implementation.



(a) AlexNet



(b) Inception V2



(c) ResNet-50

Fig. 6.4 Test accuracy of evaluated training strategies.

6.6.1 Static and Dynamic Schemes

We consider two training techniques: **static schemes** and **dynamic schemes**. When using static schemes, training uses the same data representation form for a given parameter during its complete execution. For example, Figures 6.1a and 6.1b display results obtained using static schemes. We employ the following schemes:

- **MP:** This scheme replicates prior work on MP. FMA instructions that belong to WU calculations and BN layers always use FP32 precision. The remaining FMA instructions use the MP approach represented in Figure 5.2b.
- **BF16:** FMA instructions that belong to WU calculations and BN layers always use FP32 precision. The remaining FMA instructions use BF16 operands to multiply and to accumulate (Figure 5.2c).

The *Dynamic* scheme we propose in this paper switches between the MP and BF16 static techniques during training, as explained in Section 6.4 and detailed in Algorithm 1. This dynamic method aims to retain MP's favorable training convergence properties while relying on BF16 FMAs for much of the execution.

To generate the results we show in Sections 6.7.1, 6.7.2, and 6.7.4 we set parameters $emaThreshold$, $numBatchesBF16$, and $numBatchesMP$ to 0.04, 1000, and 10, respectively, when training AlexNet, Inception V2, and ResNet-50. The *seq2seq* model employs a different data set, Multi30K, which requires adapting $emaThreshold$, $numBatchesBF16$ and $numBatchesMP$ to the number of batches per epoch of the Multi30K training process. We set them to 0.06, 15 and 1, respectively. In addition, we run an experimental campaign considering different parameter configurations in Section 6.7.3.

6.6.2 Emulating BF16

There is no real hardware supporting full BF16 FMAs, which Figure 5.2c represents. To evaluate the numerical behavior of BF16 FMAs, we use our FASE tool, described in Chapter 4.

Table 6.2 Sniper Parameters

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

Table 6.3 Accuracy and percentage of FMAs executed in *BF16* precision.

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

To assess performance improvement when using *Dynamic*, we attach the Sniper simulator [10] to FASE. Since both tools are based on Pin, combining them is natural.

Sniper is a high-speed and accurate x86 computer architecture simulator. We extend Sniper to support the AVX512 ISA, including its FMA instructions.

We simulate a standard Xeon processor by considering the hardware parameters displayed by Table 6.2. We simulate with Sniper the execution of one training batch of ResNet-50 and *seq2seq* using FP32, MP, and BF16 FMAs. Since FASE provides the percentage of batches computed in BF16 and MP during the whole training, we extrapolate the overall performance using the performance metrics that Sniper provides. This approach is equivalent to the widely used SimPoints method [32] and makes it possible to estimate the performance of the whole training workload.

6.7 Evaluation

6.7.1 Convolutional Neural Networks

Figure 6.5 shows the validation accuracy of the three considered CNN models for the different training strategies. The *x-axis* represents the epochs of the training process while the *y-axis* shows the accuracy reached by the model over the validation set. In addition, Table 6.3

shows the *Top-1* and *Top-5* validation accuracies reached on the three network models for each training strategy, along with the percentage of FMA instructions fully computed with BF16 operands. We consider the FP32, MP, and BF16 static strategies and our *Dynamic*.

Figure 6.4a shows that BF16 displays worse accuracy than *Dynamic* or MP. Table 6.3 shows that FP32, MP, *Dynamic*, and BF16 reach *Top-5* accuracies of 84.50%, 84.43%, 84.02% and 82.56% respectively after 32 epochs. Importantly, *Dynamic* reaches comparable accuracy with respect to FP32 and MP while performing 94.60% of the FMA instructions in full BF16 precision. In contrast, BF16 training does 99.93% of the FMAs in full BF16 precision (0.07% are in WU and BN layers), but the accuracy drops by almost 3% in *Top-1* and 2% in *Top-5*. This significant drop in accuracy takes place by performing an additional 5% of BF16 FMAs compared to *Dynamic*. This shows that our proposal successfully achieves SoA accuracy levels while still relying mostly on BF16 FMAs.

Figure 6.4b shows validation accuracy over 16 epochs for the Inception V2 model. Accuracy fluctuates initially due to its structure and recommended hyperparameters. *Dynamic* responds in a robust way to these changes, which highlights its general applicability. Table 6.3 shows that FP32, MP, *Dynamic*, and BF16 reach *Top-5* accuracies of 92.36%, 92.67%, 92.02%, and 92.05% respectively after 16 epochs. BF16 training is able to achieve SoA accuracy since this network is designed to be robust to noise and tolerates lower precision.

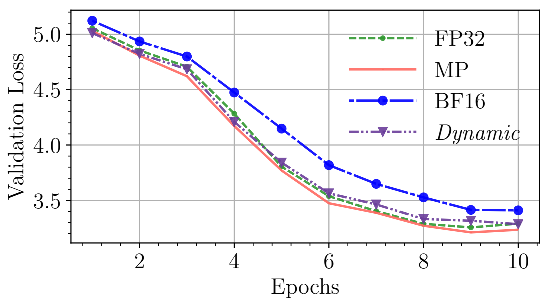
The evaluation on ResNet-50 (Figure 6.4c) demonstrates that *Dynamic* training is effective when applied to deeper CNNs. In this case, the accuracy of the model reaches SoA levels while using BF16 for 96.40% of the FMA instructions. Table 6.3 displays the accuracy numbers we obtain from our evaluation after 32 epochs. BF16 training fails to deliver SoA *Top-1* accuracy with a drop of 2.99% with respect to FP32. *Dynamic* is able to close the accuracy gap between FP32 and BF16 training by performing a large percentage of FMA instructions in BF16 precision.

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

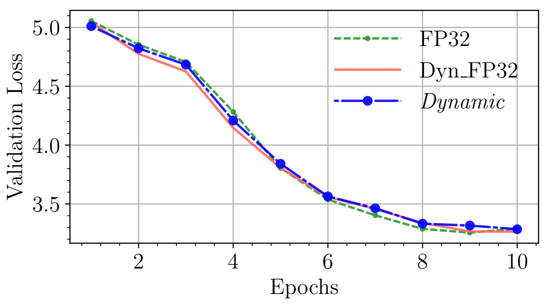
6.7.2 Sequence to Sequence RNN Model

Figure 6.5a shows the *seq2seq* validation loss achieved over 10 epochs for all the considered strategies. Table 6.4 contains the final numbers for training and validation loss, and the percentage of BF16 precision FMAs.

While FP32, MP, and *Dynamic* perform similarly for *seq2seq*, BF16 is not able to yield comparable validation loss. Table 6.4 shows that *Dynamic* achieves SoA accuracy in terms of both validation loss (Val-Loss) and training loss (Tr-Loss) while performing 66.0% of the FMA instructions fully using BF16. In addition, the loss function values for validation data



(a) Val. Loss of the *seq2seq*.



(b) Val. Loss of *Dyn_FP32*

Fig. 6.5 Validation Losses.

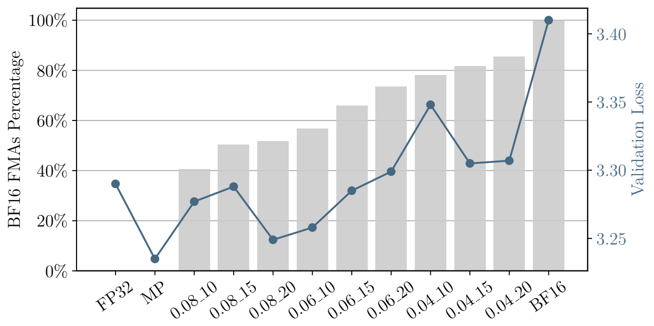


Fig. 6.6 Sensitivity of *seq2seq*.

slightly improve in MP and *Dynamic* with respect to FP32. This behavior has been studied [4].

6.7.3 Sensitivity Analysis for Dynamic Precision Algorithm

We perform a sensitivity analysis of the parameters employed in Algorithm 1. We consider *numBatchesBF16* = {500, 1000, 2000}, and *emaThreshold* = {0.02, 0.04, 0.08}, while *numBatchesMP* is set to 10. Figure 6.7 shows, for a number of ResNet-50 epochs, the accuracy obtained for each of the 9 tested configurations. In addition, we include the accuracy values of BF16, MP, and FP32. The accuracy of all *Dynamic* configurations is above BF16. The most relevant parameter is *emaThreshold*, as it decides when to switch between different FMA approaches. As long as this parameter is reasonably set to detect training loss improvement or degradation, *Dynamic* achieves SoA accuracy.

Figure 6.6 shows the sensitivity analysis for the *seq2seq* model. We consider *numBatchesBF16* = {10, 15, 20}, and *emaThreshold* = {0.04, 0.06, 0.08}, while *numBatchesMP* is set to 1. Bars show the percentage of FMAs run using BF16 format per each parameter configuration, and the line represents its corresponding validation loss. We find that validation loss improves the FP32 value for some configurations. This is due to the numerical noise injected

Table 6.4 Loss and FMAs executed in BF16 precision for *seq2seq*

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

Table 6.5 Performance results

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

by reduced data representation formats, which may help the training processes in some scenarios. Previous work has also observed this effect [4]. Multiple *Dynamic* training configurations obtain SoA validation loss with up to 66% BF16 FMAs.

6.7.4 Dynamically Switching Between FP32 and BF16

The evaluation of Sections 6.7.1, 6.7.2, and 6.7.3 considers a *Dynamic* approach that switches between MP and BF16 FMAs. However, the *Dynamic* technique can also switch between FP32 and BF16. Figure 6.5b shows validation loss evolution while training *seq2seq* considering a *Dyn_FP32* approach, which switches between FP32 and BF16. Results obtained with *Dyn_FP32* do not present any noticeable deviation to the ones obtained with *Dynamic* in terms of accuracy and percentage of BF16 FMA instructions. Similar behavior is observed with the CNN models. This certifies that our proposal can be applied in machines that support FP32 and BF16 FMAs, without needing MP.

6.7.5 Sniper Results

We evaluate the performance of BF16 training by attaching the FASE tool to the Sniper simulator. Table 7.5 shows the training time of one batch using FP32, MP, and full BF16 FMAs. In addition, we show the speed-up of *Dynamic* (MP-BF16) and *Dyn_FP32* (FP32-BF16) w.r.t FP32 training. We use ResNet-50 and *seq2seq* as example models.

As can be seen in the table, the execution time of one batch using BF16 is significantly faster than FP32 or MP. This is due to lower memory bandwidth requirements and doubling the FMA vectorization (AVX512) throughput, as having all input and output operands in 16

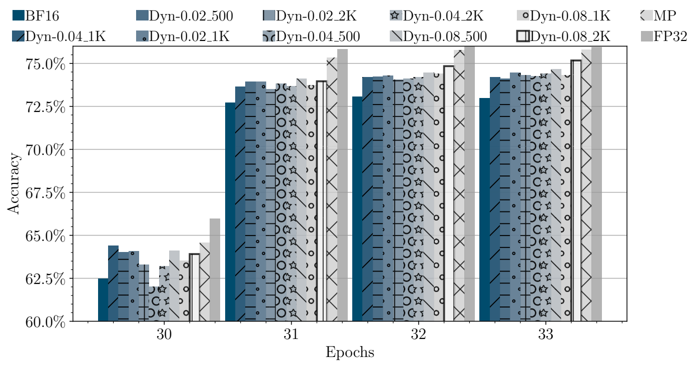


Fig. 6.7 Sensitivity analysis of *Dynamic* on ResNet-50.

bits enables more data level parallelism. The speed-ups achieved for the entire training process with the *Dynamic* approach reach 1.39× and 1.26× for ResNet-50 and *seq2seq*, respectively.

6.8 Conclusions

We analyze the instruction mix of DNN training workloads. We show that FMA instructions represent 60% of these workloads. While MP training can deliver SoA accuracy, training schemes that rely on half-precision FMAs, like BF16 training, fail to deliver comparable accuracy levels. We propose a *Dynamic* training technique that can perform a large portion of FMAs in full half-precision, lowering training requirements without hurting accuracy. We achieve this by training in BF16 mode and identifying when training convergence stagnates, at which point we switch to a higher precision strategy like MP or FP32 until stagnation dissipates [72].

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

Chapter 7

A BF16 FMA is All You Need for DNN Training

7.1 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 [90]. 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 features 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 [35, 91, 96]. This advantage in terms of area budget has motivated the adoption of the BF16 format, which features 1 sign, 8 exponent, and 7 explicit mantissa bits, by many hardware vendors [35, 68]. 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 [35, 68].

Despite these advantages, any proposal does not fully exploit the potential of using BF16 during the entire training process. Previous work has described the numerical issues of BF16, which are caused by its reduced mantissa bits budget [51, 64]. To overcome them, previous approaches combine the BF16 format with the standard IEEE 754 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 WU operations or BN must entirely rely on the FP32 format to achieve state-of-the-art accuracy [51, 64].

There are more aggressive proposals that use 4-bit (FP4) datatypes for ultra-low precision training [85] or that dynamically employ between 3 and 8 bits of precision [25, 26]. 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.

We propose 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.

We use FASE to evaluate the numerical behavior of $\text{FMA}^{\text{bf16}}_{\text{n\_m}}$. FASE uses Intel Pin [61] to intercept FMA instructions and modify its floating-point operands. Using FASE we evaluate 7 $\text{FMA}^{\text{bf16}}_{\text{n\_m}}$ variants on a wide range of DNNs: ResNet (18, 34, 50, 101) [34] and MobileNetV2 [81] on the CIFAR10 and CIFAR100 datasets, LSTMx2 model on Penn Treebank (PTB) dataset [102], two transformer-based models using IWSLT16 [94] and Multi30K dataset [23, 93], and the Neural Graph Collaborative Filtering (NGCF) [98] a recommender system applied to the MovieLens: ML-100k dataset [33]. 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 using Snipersim [10] to evaluate the performance of $\text{FMA}^{\text{bf16}}_{\text{n\_m}}$ operators when replacing 32-bit FMA functional units on a Skylake-like processor. Our evaluation shows that $\text{FMA}^{\text{bf16}}_{\text{1\_2}}$ and $\text{FMA}^{\text{bf16}}_{\text{2\_2}}$ reach $1.34\times$ and $1.28\times$ speed-up on ResNet101, respectively, with respect to FP32 at equivalent area.

To the best of our knowledge, $\text{FMA}^{\text{bf16}}_{\text{n\_m}}$ 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$\times$ 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.

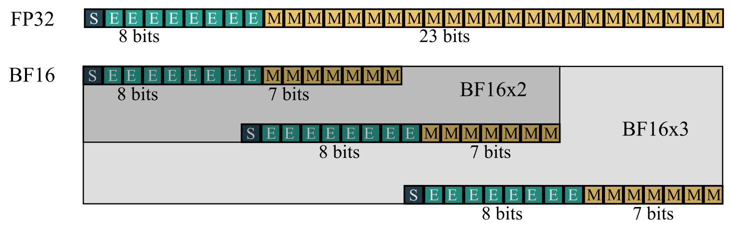


Fig. 7.1 BF16xN Data Formats.

7.2 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. [35] 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 7.1, 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 chapter, 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 7.1, or to its BF16X3 compound datatype representation using the three equations of Formula 7.1 [35]. 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.

$$\begin{aligned} a_0 &= BF(a) \\ a_1 &= BF(a - a_0) \\ a_2 &= BF(a - (a_0 - a_1)) \end{aligned} \tag{7.1}$$

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 $+/-Inf$.

7.2.1 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 7.2 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 [35].

$$\begin{aligned} c := c + a \cdot b = c + a_0 \cdot b_0 + a_0 \cdot b_1 + a_0 \cdot b_2 \\ + a_1 \cdot b_0 + a_1 \cdot b_1 + a_2 \cdot b_0 \\ + a_1 \cdot b_2 + a_2 \cdot b_1 + a_2 \cdot b_2 \end{aligned} \tag{7.2}$$

Henry et al. [35] 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 7.3) can be avoided, while providing accuracy within the $[23,24]$ bit range.

$$\begin{aligned} c := c + a \cdot b \approx c + a_0 \cdot b_0 + a_0 \cdot b_1 + a_0 \cdot b_2 \\ + a_1 \cdot b_0 + a_1 \cdot b_1 + a_2 \cdot b_0 \end{aligned} \tag{7.3}$$

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 7.4 shows.

$$\begin{aligned} a \cdot b = (a_0 + a_1) \cdot (b_0 + b_1) \\ a \cdot b = a_0 \cdot b_0 + a_0 \cdot b_1 \\ a_1 \cdot b_0 + a_1 \cdot b_1 \end{aligned} \tag{7.4}$$

Following a similar reasoning as Henry et al. [35] with BF16x3, we can drop the last

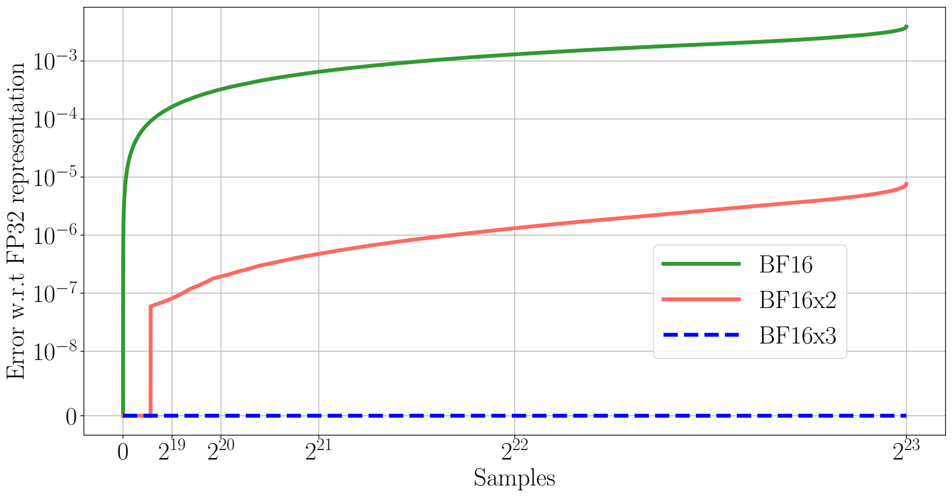


Fig. 7.2 Error with respect to FP32 representation for different BF16xN formats for all possible *mantissa* combinations over a fixed exponent (2^{23} samples).

term, $a_1 \cdot b_1$ while still keeping $[15, 16]$ bits of mantissa accuracy for a wide range of values.

Henry et al. [35] require FP32 support for the accumulator input c, and for the conversion between FP32 and BF16xN. Our proposed $\text{FMA}^{\text{bf16}}_{\text{n\_m}}$ operators, introduced in Section 7.4, do not require FP32 support as all input and output operands are expressed as BF16 literals.

7.3 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 WU or BN.

7.3.1 BF16xN Representation Errors

Figure 7.2 shows numerical errors of BF16x1, BF16x2, and BF16x3 FMAs when representing all the 2^{23} possible representations of FP32 explicit mantissa bits. Without loss 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 [51, 64]. 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 7.2 shows how the BF16x2 error is consistently 3 orders of magnitude

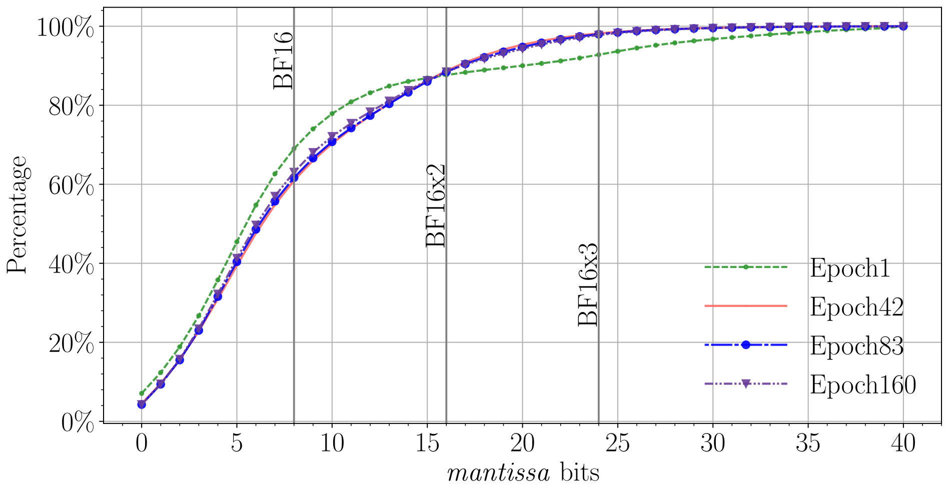


Fig. 7.3 Percentage of FMAs without *swamping* for a given number of *mantissa* bits during ResNet101 training

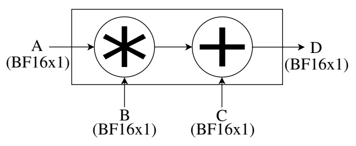
lower than BF16X1. As Section 7.7 demonstrates, this low error enables full DNN training with the BF16X2 datatype. Finally, the BF16X3 format has no significant error compared to FP32. Indeed, it has enough *mantissa* bits to exactly represent all FP32 samples.

7.3.2 *Swamping* Issues in DNN Training

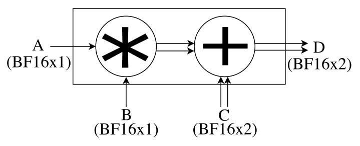
During the accumulation phase of FMA instructions, the *mantissa* of the smallest value shifts according to the exponent difference between the operands. This shift brings the possibility to eliminate the smallest operand when the exponent difference is bigger than the number of *mantissa* bits. This full absorption issue is known as *swamping* [96] in the deep learning community. In Section 6.3.3 we use ResNet50 model to explain this issue.

Figure 7.3 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 7.6 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.

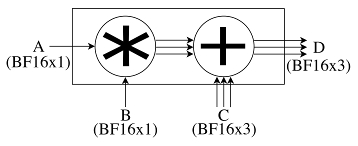
Diminishing the number of *mantissa* bits makes DNNs more sensitive to *swamping* and may lead to critical information loss. In fact, the use of BF16X1 leads to a substantial amount of *swamping* that prevents reaching state-of-the-art training accuracy levels [96]. The BF16X2 and BF16X3 data formats entirely rely on BF16 arithmetic functional units and do not suffer from the common numerical issues of BF16X1 that Figures 7.2 and 7.3 display.



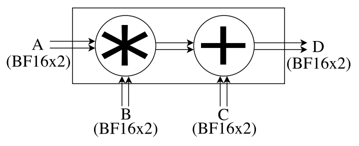
(a) $\mathrm{FMA}^{\mathrm{bf16}}_{1\_1}$



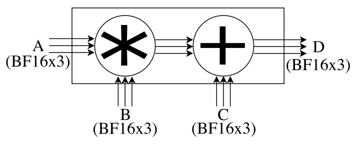
(b) $\mathrm{FMA}^{\mathrm{bf16}}_{1\_2}$



(c) $\mathrm{FMA}^{\mathrm{bf16}}_{1\_3}$



(d) $\mathrm{FMA}^{\mathrm{bf16}}_{2\_2}$



(e) $\mathrm{FMA}^{\mathrm{bf16}}_{3\_3}$

Fig. 7.4 $\mathrm{FMA}^{\mathrm{bf16}}_{\mathrm{n\_m}}$ Operators

Table 7.1 $\mathrm{FMA}^{\mathrm{bf16}}_{\mathrm{n\_m}}$ 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 the equivalent area.

$\mathrm{FMA}^{\mathrm{bf16}}_{\mathrm{n\_m}}$	$\mathrm{FMA}^{\mathrm{bf16}}_{1\_1}$	$\mathrm{FMA}^{\mathrm{bf16}}_{1\_2}$	$\mathrm{FMA}^{\mathrm{bf16}}_{1\_3}$	$\mathrm{FMA}^{\mathrm{bf16}}_{2\_2}$ {3}	$\mathrm{FMA}^{\mathrm{bf16}}_{2\_2}$ {4}	$\mathrm{FMA}^{\mathrm{bf16}}_{3\_3}$ {6}	$\mathrm{FMA}^{\mathrm{bf16}}_{3\_3}$ {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 $> 2^{-103}$

7.4 FMA Operators Based on BF16 Arithmetic

This section introduces a new class of FMA operators, $\mathrm{FMA}^{\mathrm{bf16}}_{\mathrm{n\_m}}$, that entirely rely on BF16-based operands. When performing an FMA operation $D = A \cdot B + C$, $\mathrm{FMA}^{\mathrm{bf16}}_{\mathrm{n\_m}}$ 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.

7.4.1 The $\mathrm{FMA}^{\mathrm{bf16}}_{\mathrm{n\_m}}$ operators

We propose and evaluate four new $\mathrm{FMA}^{\mathrm{bf16}}_{\mathrm{n\_m}}$ operators: $\mathrm{FMA}^{\mathrm{bf16}}_{1\_2}$, $\mathrm{FMA}^{\mathrm{bf16}}_{1\_3}$, $\mathrm{FMA}^{\mathrm{bf16}}_{2\_2}$, and $\mathrm{FMA}^{\mathrm{bf16}}_{3\_3}$. The first two operators, which are represented in Figures 7.4b and 7.4c, 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 7.4d) and $\text{FMA}^{\text{bf16}}_{\text{3\_3}}$ (Figure 7.4e). As Section 7.2 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 7.5 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 7.3 and 7.2 indicate. We call these operators $\text{FMA}^{\text{bf16}}_{\text{3\_3}}$ $\{6\}$ and $\text{FMA}^{\text{bf16}}_{\text{3\_3}}$ $\{9\}$.

$$\begin{aligned} c = c + a \cdot b \approx (c_0 + c_1) + a_0 \cdot b_0 + a_0 \cdot b_1 \\ a_1 \cdot b_0 + a_1 \cdot b_1 \end{aligned} \tag{7.5}$$

Figure 7.4 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 Equation 7.4, 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 7.4. This 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 7.7 demonstrates that $\text{FMA}^{\text{bf16}}_{\text{n\_m}}$ operators achieve comparable accuracy with respect to FP32 executions on large and complex DNNs. Our $\text{FMA}^{\text{bf16}}_{\text{n\_m}}$ operators entirely use BF16 arithmetic, that is, they do not employ FP32 computations on any layer, including BN, Softmax, and WU routines.

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

To characterize our $\text{FMA}^{\text{bf16}}_{\text{n\_m}}$ units we use the observation that the area of an FMA is dominated by the multiplier as it grows quadratically with mantissa bits [91] [96]. 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 [69] or Google's TPU [97]) leads to efficient matrix compute engines that can deliver 8-32× more FLOP/S than equivalent FP32 hardware [35]. 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 7.1 characterizes $\mathrm{FMA}^{\mathrm{bf16}}_{\mathrm{n\_m}}$ 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 $\mathrm{FMA}^{\mathrm{bf16}}_{\mathrm{n\_m}}$ 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 $\mathrm{FMA}^{\mathrm{bf16}}_{\mathrm{1\_1}}$, $\mathrm{FMA}^{\mathrm{bf16}}_{\mathrm{1\_2}}$ and $\mathrm{FMA}^{\mathrm{bf16}}_{\mathrm{1\_3}}$ 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 $\mathrm{FMA}^{\mathrm{bf16}}_{\mathrm{2\_2}}$ {3} and $\mathrm{FMA}^{\mathrm{bf16}}_{\mathrm{2\_2}}$ {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 $\mathrm{FMA}^{\mathrm{bf16}}_{\mathrm{3\_3}}$ {6} and $\mathrm{FMA}^{\mathrm{bf16}}_{\mathrm{3\_3}}$ {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 7.1 characterizes $\mathrm{FMA}^{\mathrm{bf16}}_{\mathrm{n\_m}}$ 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 7.7 provides performance in terms of total elapsed time with respect to FP32 using an accurate micro-architectural simulator.

7.5 FASE: An $\mathrm{FMA}^{\mathrm{bf16}}_{\mathrm{n\_m}}$ Emulation Tool

To assess the numerical properties of DNN training workloads when they are exposed to the $\mathrm{FMA}^{\mathrm{bf16}}_{\mathrm{n\_m}}$ units, we rely on FASE. There is no hardware supporting FMA units relying entirely 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) [41, 66, 97]. Alternatively, [12] and [51] 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 [43] is not possible for most users as the code is not publicly available.

FASE can detect, intercept, and instrument any instruction appearing in the program control flow, including those coming from dynamically linked libraries.

In particular, FASE dynamically changes the value of each FMA input and output operand to implement our custom $\mathrm{FMA}^{\mathrm{bf16}}_{\mathrm{n\_m}}$ arithmetic. When the BF16X3 datatype is used as output, converting the output back to the native type (FP32) to store the result in the destination register can introduce rounding noise; however, the resulting relative error is below 2^{-24} and it does not impact the accuracy of $\text{FMA}^{\text{bf16}}_{\text{n\_m}}$ arithmetic. Operands that do not use BF16x3 type are not affected by this rounding error. An advantage of FASE is its precise computer arithmetic emulation at the register level. In contrast, other approaches based on software modifications might suffer from numerical artifacts and wrong interpretations due to compiler optimizations [19].

FASE performs the following steps to emulate $\text{FMA}^{\text{bf16}}_{\text{n\_m}}$ operators when computing the FMA instruction $D = A \cdot B + C$:

- To intercept each FMA instruction during execution.
- To convert each FMA FP32 input operand to the appropriate representation using Equation 7.1, i.e., BF16x1, BF16x2 or BF16x3, depending on the currently evaluated $\text{FMA}^{\text{bf16}}_{\text{n\_m}}$ operator.
- To perform the FMA multiplication step. We employ partial products described in Equations 7.2, 7.3 and 7.4, depending on the specific $\text{FMA}^{\text{bf16}}_{\text{n\_m}}$ operator.
- To accumulate the output of $A \cdot B$ with input C.
- To update the destination register. The output is converted back to FP32 to match the register type.

FASE seamlessly works on frameworks such as Tensorflow, PyTorch or Caffe without requiring any source code modification.

7.6 Experimental Methodology

Our experimental methodology considers the $\text{FMA}^{\text{bf16}}_{\text{1\_1}}$, $\text{FMA}^{\text{bf16}}_{\text{1\_2}}$, $\text{FMA}^{\text{bf16}}_{\text{1\_3}}$, $\text{FMA}^{\text{bf16}}_{\text{2\_2}}$ $\{3\}$, $\text{FMA}^{\text{bf16}}_{\text{2\_2}}$ $\{4\}$, $\text{FMA}^{\text{bf16}}_{\text{3\_3}}$ $\{6\}$, and $\text{FMA}^{\text{bf16}}_{\text{3\_3}}$ $\{9\}$ operators. In contrast with prior proposals [51, 64], we do not employ FP32 precision on any routines to improve training convergence. Therefore, when using $\text{FMA}^{\text{bf16}}_{\text{n\_m}}$ operators, all FMA instructions use BF16 arithmetic for the whole training process. We compare the training accuracy of $\text{FMA}^{\text{bf16}}_{\text{n\_m}}$ approaches against two baselines: A full FP32 training and an approach that uses mixed-precision FMAs, described in Section 7.6.3. 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).

We test our approaches taking into account the models explained in Chapter 3

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

To train and validate ResNet, LSTMx2, the transformer-based models, and the recommender system we use a source code compiled version of PyTorch [76] (version 1.8.0), Intel MKLDNN [39] (version 1.22.0) and the Intel MKL library [43] (version 2019.4).

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

We evaluate the impact of using $\mathrm{FMA}^{\mathrm{bf16}}_{\mathrm{n\_m}}$ operators in terms of performance using the Snipersim micro-architectural simulator [10]. While FASE enables highly accurate analysis of the ability of $\mathrm{FMA}^{\mathrm{bf16}}_{\mathrm{n\_m}}$ 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 $\mathrm{FMA}^{\mathrm{bf16}}_{\mathrm{n\_m}}$. Both Sniper and FASE use Intel Pin 3.7 [61], which enables 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 7.2 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 $\mathrm{FMA}^{\mathrm{bf16}}_{\mathrm{1\_1}}$, $\mathrm{FMA}^{\mathrm{bf16}}_{\mathrm{1\_2}}$, and $\mathrm{FMA}^{\mathrm{bf16}}_{\mathrm{2\_2}}$ $\{4\}$ operators.

Since all $\mathrm{FMA}^{\mathrm{bf16}}_{\mathrm{n\_m}}$ operators rely on BF16 arithmetic, our AVX512 model assumes BF16 support per each 16-bit lane. To simulate the additional throughput achievable for each $\mathrm{FMA}^{\mathrm{bf16}}_{\mathrm{n\_m}}$ operator at equivalent FP32 area, we model wider functional units using the numbers from Table 7.1 and the explanation in Section 7.4.2. 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.

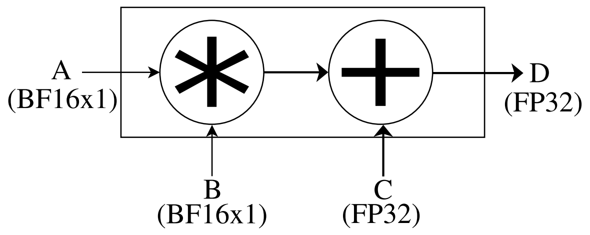


Fig. 7.5 Mixed Precision *Fused Multiply-Add* (FMA).

Table 7.2 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

7.6.3 Mixed-Precision FMA

Our evaluation considers the Mixed-Precision (MP) FMA instruction used by the Intel Advanced Matrix Extensions (Intel AMX) [42] or the Nvidia Ampere architecture [68]. Figure 7.5 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* [96]. MP training also employs full FP32 FMAs to process BN and WU layers.

7.7 Evaluation

We evaluate the training accuracy of the $\text{FMA}^{\text{bf16}}_{1\_1}$, $\text{FMA}^{\text{bf16}}_{1\_2}$, $\text{FMA}^{\text{bf16}}_{1\_3}$, $\text{FMA}^{\text{bf16}}_{2\_2}$ $\{3\}$, $\text{FMA}^{\text{bf16}}_{2\_2}$ $\{4\}$, $\text{FMA}^{\text{bf16}}_{3\_3}$ $\{6\}$, and $\text{FMA}^{\text{bf16}}_{3\_3}$ $\{9\}$ operators when training object classification and natural language processing models in Sections 7.7.1 and 7.7.2, respectively. We also consider FP32 and MP, which we describe in Section 7.6. Section 7.7.4 evaluates the performance improvements of $\text{FMA}^{\text{bf16}}_{n\_m}$ operators with respect to FP32.

7.7.1 Training Accuracy: Object Classification

Figure 7.6 shows the MobileNetV2 validation accuracy on CIFAR100 when using FP32, MP and seven $\text{FMA}^{\text{bf16}}_{n\_m}$ 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, $\text{FMA}^{\text{bf16}}_{1\_2}$ and $\text{FMA}^{\text{bf16}}_{1\_1}$ obtain accuracies of 75.04%, 75.16%, 74.85% and 73.92%, respectively. The $\text{FMA}^{\text{bf16}}_{1\_1}$ approach fails to deliver similar accuracy as FP32. In contrast, $\text{FMA}^{\text{bf16}}_{1\_2}$ outperforms $\text{FMA}^{\text{bf16}}_{1\_1}$ and reaches similar accuracy as FP32 and MP. Higher precision operators like $\text{FMA}^{\text{bf16}}_{2\_2}$ $\{3\}$ or $\text{FMA}^{\text{bf16}}_{3\_3}$ $\{6\}$ reach 74.82% and 75.31% accuracy, respectively. They obtain similar or better accuracy than $\mathrm{FMA}_{1\_2}^{\mathrm{bf16}}$.

Figure 7.7 shows our evaluation results considering the ResNet18 model using the CIFAR100 dataset. $\mathrm{FMA}_{1\_1}^{\mathrm{bf16}}$ 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 $\mathrm{FMA}_{\mathrm{n\_m}}^{\mathrm{bf16}}$ versions to be trained. Figure 7.8 shows results for ResNet34. In this case, the $\mathrm{FMA}_{1\_1}^{\mathrm{bf16}}$ approach loses almost 1.0% compared to FP32, while $\mathrm{FMA}_{1\_2}^{\mathrm{bf16}}$ 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 $\mathrm{FMA}_{\mathrm{n\_m}}^{\mathrm{bf16}}$ 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 $\mathrm{FMA}_{\mathrm{n\_m}}^{\mathrm{bf16}}$ approaches achieve when training the ResNet50 and ResNet101 models, which are displayed in Figures 7.9 and 7.10, respectively. $\mathrm{FMA}_{1\_1}^{\mathrm{bf16}}$ accuracy is significantly lower than FP32.

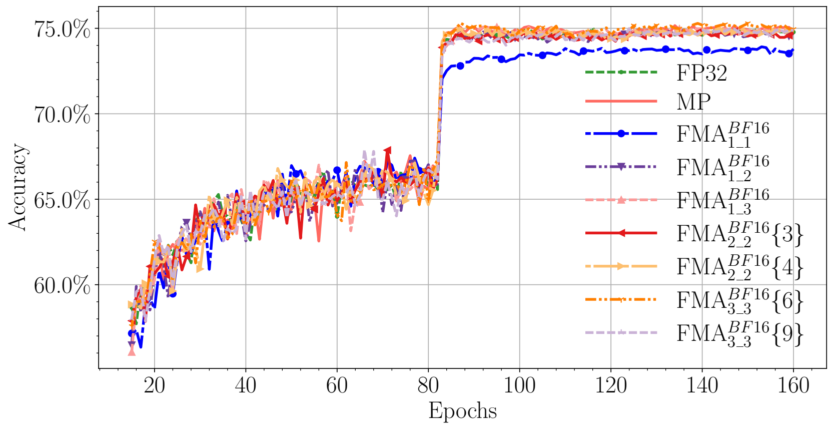


Fig. 7.6 MobilenetV2 Accuracy on CIFAR100 Validation Set.

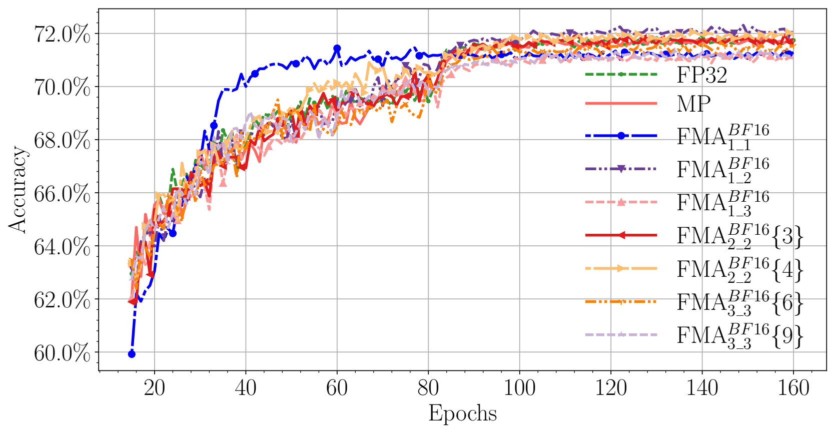


Fig. 7.7 ResNet18 Accuracy on CIFAR100 Validation Set.

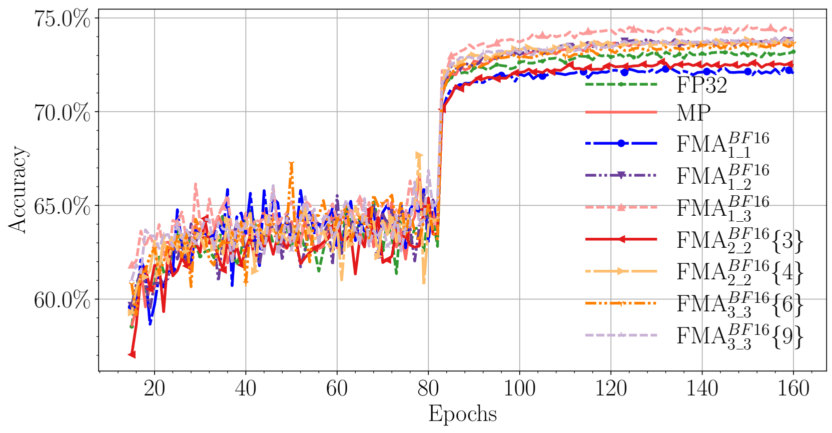


Fig. 7.8 ResNet34 Accuracy on CIFAR100 Validation Set.

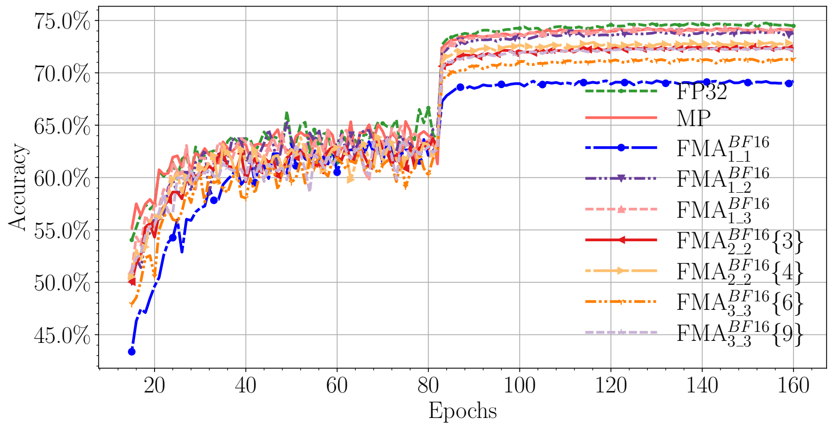


Fig. 7.9 ResNet50 Accuracy on CIFAR100 Validation Set.

In ResNet50 $\mathrm{FMA}_{1\_2}^{\mathrm{bf16}}$ already achieves similar accuracy when compared to FP32 and MP. In contrast, ResNet101 $\mathrm{FMA}_{1\_2}^{\mathrm{bf16}}$ does have a slight drop with respect to FP32, from 75.93% to 73.75%, respectively. However, the use of $\mathrm{FMA}_{2\_2}^{\mathrm{bf16}}$ {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 7.11 shows the results of training ResNet34 using the CIFAR10 dataset, again the trend is that operators using more accumulator bits obtain better accuracy results. In this specific case, $\mathrm{FMA}_{1\_2}^{\mathrm{bf16}}$ attains a validation accuracy of 93.86%, which is comparable to that achieved by FP32 (94.30%). In ResNet50 with CIFAR10, Figure 7.12, $\mathrm{FMA}_{1\_3}^{\mathrm{bf16}}$ 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 $\mathrm{FMA}_{2\_2}^{\mathrm{bf16}}$ operator displays the same levels of accuracy (94.05%) with better trade-offs in terms of storage

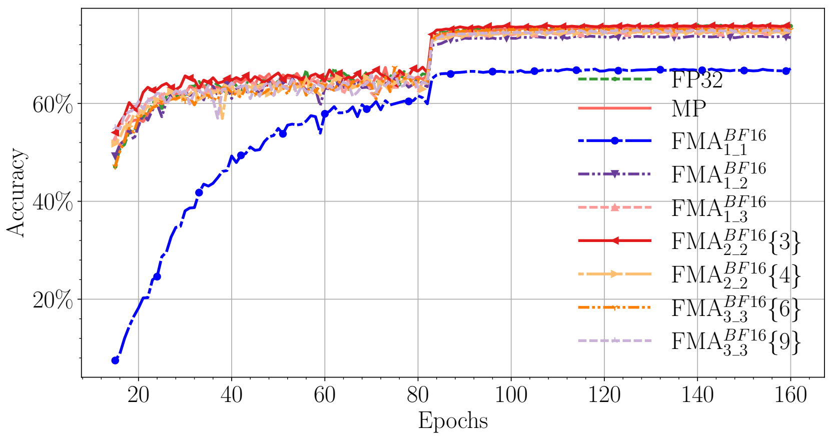


Fig. 7.10 ResNet101 Accuracy on CIFAR100 Validation Set.

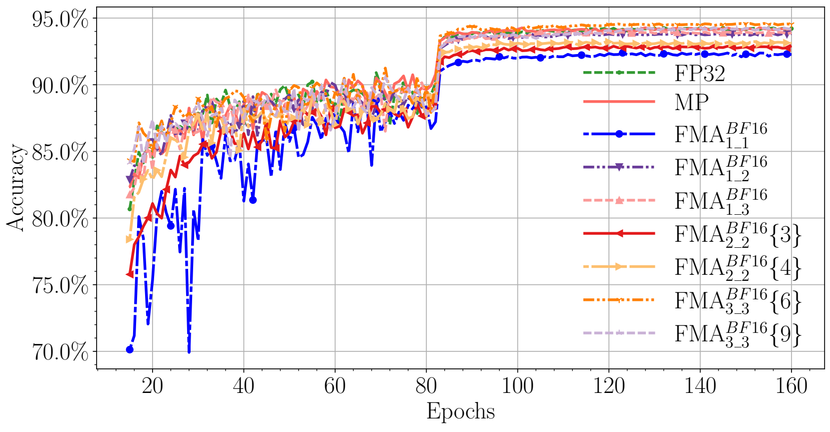


Fig. 7.11 ResNet34 Accuracy on CIFAR10 Validation Set.

and bandwidth requirements. Finally, similar results are obtained in ResNet101 (Figure 7.13), where the $FMA_{1\_2}^{bf16}$ operator gives an accuracy result of 94.31% while FP32 obtains a 94.71%.

Tables 7.3 and 7.4 summarize the maximum top1 validation accuracy achieved by FP32, MP, and the 7 $FMA_{n\_m}^{bf16}$ approaches on the 4 ResNet models and MobileNetV2 for the CIFAR10 and CIFAR100 datasets, respectively. We observe that $FMA_{1\_1}^{bf16}$ fails to deliver the same accuracy as FP32 for the deepest models, i.e., ResNet34, ResNet50, and ResNet101. $FMA_{1\_2}^{bf16}$ also achieves worse accuracy than FP32, particularly for the case of ResNet101 and the CIFAR100 data set. In contrast, the two $FMA_{2\_2}^{bf16}$ variants behave very similarly as FP32 and MP. The very deep nature of ResNet101 requires an operator like $FMA_{2\_2}^{bf16}$ $\{3\}$, which increases the representation accuracy of some FMA input parameters.

Some high-precision $FMA_{n\_m}^{bf16}$ operators sometimes behave worse than others even if they have larger numerical precision. Audhkhasi et al. [4] explain this effect where noise can speed-up convergence during backpropagation. This can be observed in ResNet50 with CIFAR100

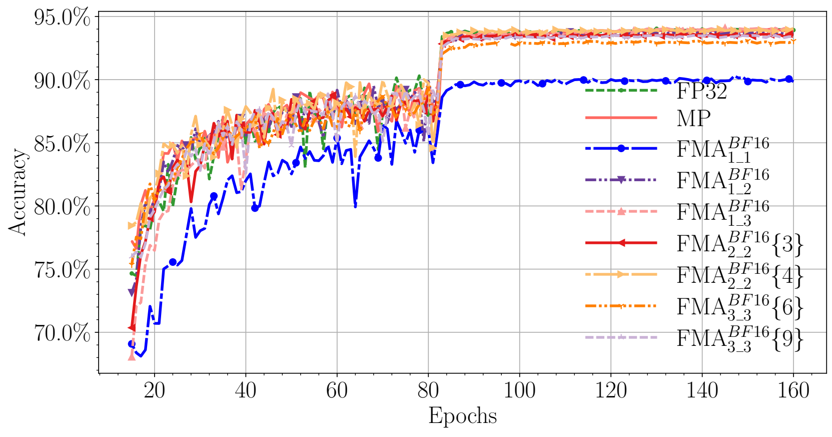


Fig. 7.12 ResNet50 Accuracy on CIFAR10 Validation Set.

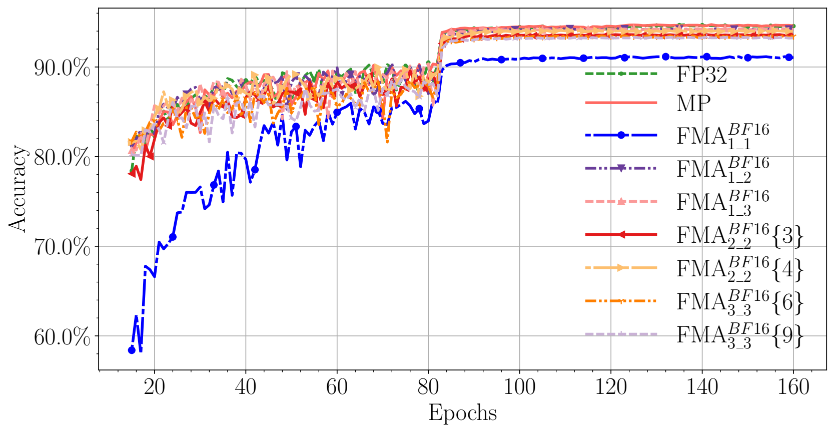


Fig. 7.13 ResNet101 Accuracy on CIFAR10 Validation Set.

where FMA$_{2\_2}^{\mathrm{bf16}}$ {4} has lower accuracy than FP32 and FMA$_{1\_2}^{\mathrm{bf16}}$; however, in ResNet101 FMA$_{2\_2}^{\mathrm{bf16}}$ {3} achieves the highest accuracy.

In conclusion, FMA$_{\mathrm{n\_m}}^{\mathrm{bf16}}$ 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$_{2\_2}^{\mathrm{bf16}}$ variants deliver comparable state-of-the-art accuracy with respect to FP32 while potentially providing better performance. Section 7.7.4 evaluates FMA$_{2\_2}^{\mathrm{bf16}}$ {4} in terms of performance.

7.7.2 Training Accuracy: Natural Language Processing

We consider three natural language processing models in this section. Figure 7.14 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$_{1\_1}^{\mathrm{bf16}}$ operator fails to converge, giving a NaN

Table 7.3 Accuracy on Validation Set for CIFAR10

Model	FP32	MP	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}
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 7.4 Accuracy on Validation Set for CIFAR100

Model	FP32	MP	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}
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%

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 7.15 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 7.14. 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 Task (NMT) using the IWSLT16 dataset. Figure 7.16 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$^{bf16}_{1\_1}$ produces state-of-the-art results [89].

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 7.17 shows that the models converge reaching the same training perplexity. We also compute the

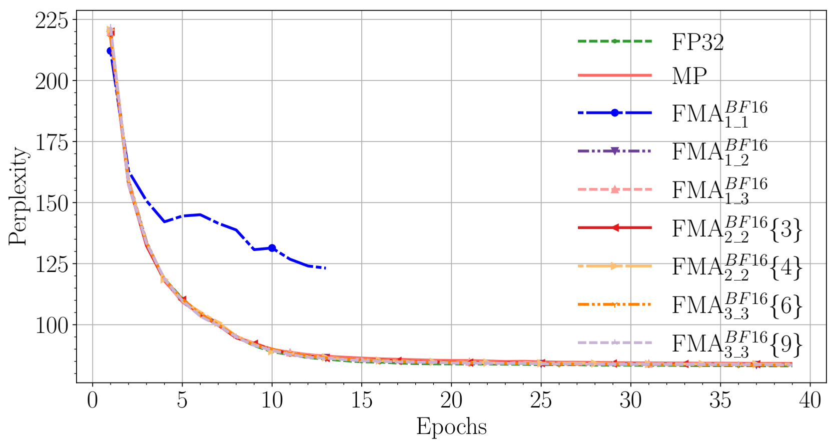


Fig. 7.14 LSTMx2 Test Perplexity on PTB dataset

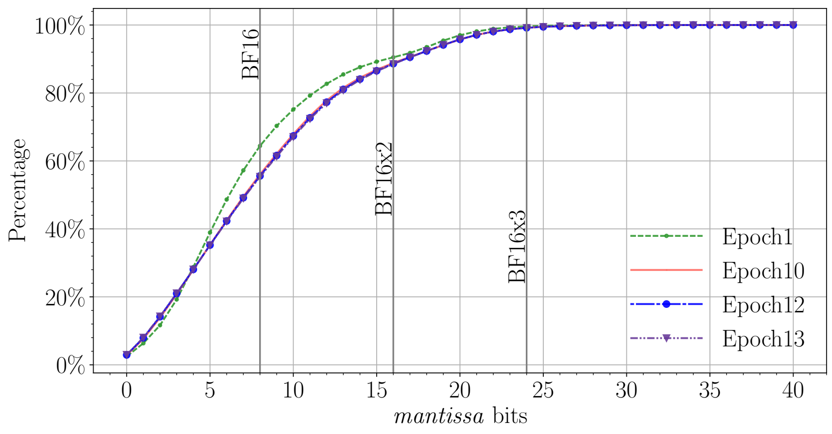


Fig. 7.15 *Swamping* analysis on LSTMx2 Model.

final BLEU Scores, which are 35.67, 36.05, 36.33 and 36.08 for FP32, $FMA^{bf16}_{1\_1}$, $FMA^{bf16}_{2\_2}$ {4} and $FMA^{bf16}_{3\_3}$ {6}, respectively. Another example that transformer-based models display robust numerical properties with BF16 low precision operators.

To explain the good performance of $FMA^{bf16}_{1\_1}$ with transformer models, we carry out the *swamping* analysis on several epochs when training IWSLT16. Figure 7.18 shows how $FMA^{bf16}_{1\_1}$ 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, $FMA^{bf16}_{n\_m}$ operators match the accuracy of 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.

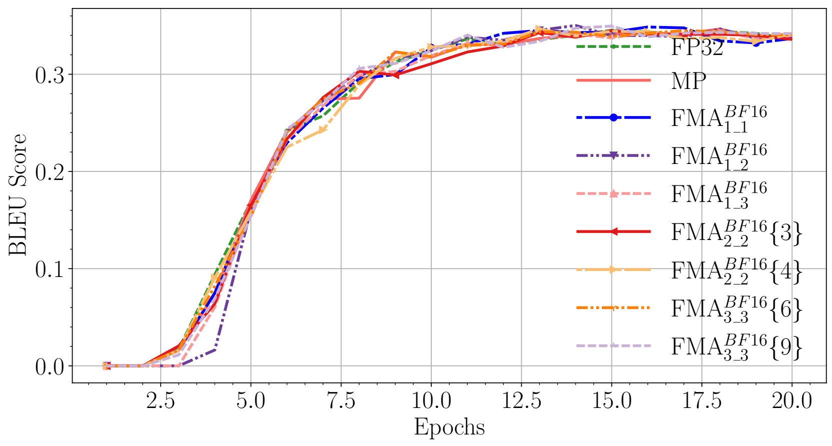


Fig. 7.16 Transformer BLEU Score on IWSLT16.

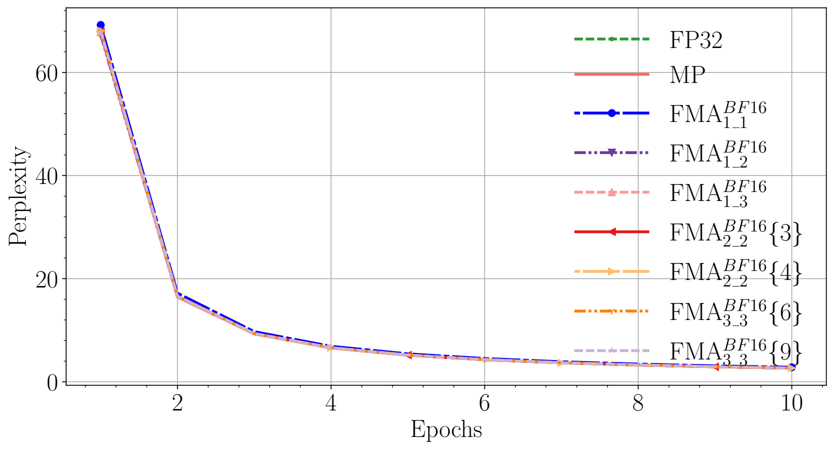


Fig. 7.17 Transformer Training Perplexity on Multi30k.

7.7.3 Training Accuracy: Recommender Systems

Figure 7.19 shows the training process of the recommender system introduced in Section 3.2.1. 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, $\mathrm{FMA}_{1\_1}^{\mathrm{bf16}}$ 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 ±0.2%. The best result is obtained by $\mathrm{FMA}_{2\_2}^{\mathrm{bf16}}$ {4} with 66.95%. We can again conclude that $\mathrm{FMA}_{\mathrm{n\_m}}^{\mathrm{bf16}}$ operators can deliver state-of-the-art accuracy.

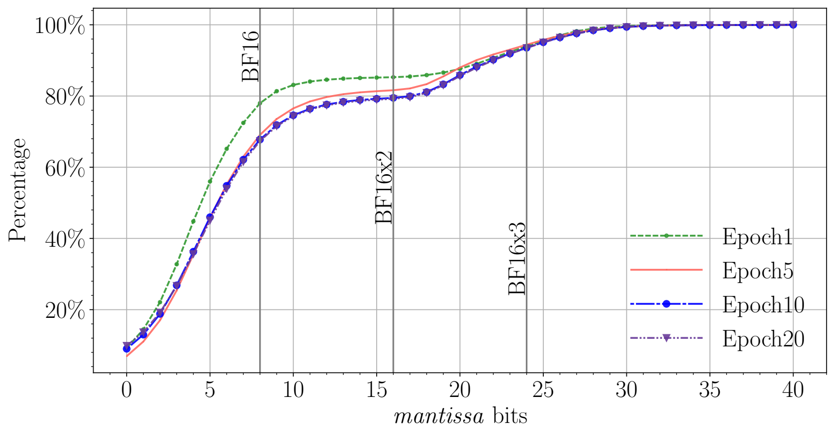


Fig. 7.18 *Swamping* analysis on Transformer model using IWSLT16 dataset.

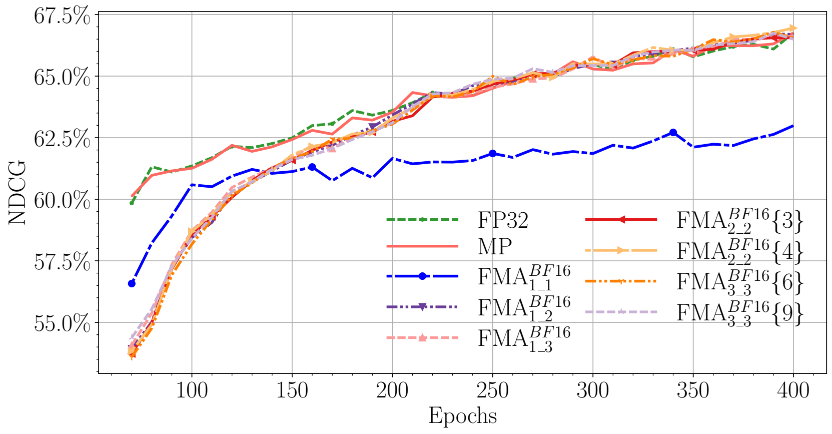


Fig. 7.19 Normal Discounted Cumulative Gain on MovieLens Dataset.

7.7.4 Performance Evaluation

We use the Sniper simulator to evaluate the performance benefits of the $FMA_{1\_1}^{bf16}$, $FMA_{1\_2}^{bf16}$, and $FMA_{2\_2}^{bf16}$ {4} operators. Table 7.5 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 $FMA_{1\_1}^{bf16}$, $FMA_{1\_2}^{bf16}$, and $FMA_{2\_2}^{bf16}$ {4} respectively. The transformer model presents similar results, while LSTMx2 shows a slight reduction in performance improvements due to an instruction mix with a lower FMA instruction count.

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 with respect to the total instruction count, the more potential $FMA_{n\_m}^{bf16}$ operators have in terms of performance improve-

Table 7.5 Performance speed-up estimations using the Sniper Simulator

Model	**One Batch**			
	FP32	$\text{FMA}^{\text{bf16}}_{\text{1\_1}}$	$\text{FMA}^{\text{bf16}}_{\text{1\_2}}$	$\text{FMA}^{\text{bf16}}_{\text{2\_2}}$ {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×

ments. 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 $\text{FMA}^{\text{bf16}}_{\text{1\_2}}$ operator accelerates FMA instructions by 2.80× with respect to FP32, which leads to the reported 1.34× for the whole execution. $\text{FMA}^{\text{bf16}}_{\text{n\_m}}$ 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 $\text{FMA}^{\text{bf16}}_{\text{n\_m}}$ 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.

7.8 Conclusion and Future Work

We propose a new class of FMA operators, $\text{FMA}^{\text{bf16}}_{\text{n\_m}}$, that rely entirely on BF16 arithmetic but can achieve FP32 accuracy through compound datatypes (BF16xN) [73]. 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 $\text{FMA}^{\text{bf16}}_{\text{n\_m}}$ operators that feature different levels of accuracy and theoretical throughput improvements at iso-area with respect to an FP32 FMA unit.

To evaluate the training accuracy of the proposed operators on a wide range of DNN workloads we develop FASE, a binary analysis tool that enables $\text{FMA}^{\text{bf16}}_{\text{n\_m}}$ emulation and seamlessly works with PyTorch, Caffe, or TensorFlow. For all the evaluated networks, the proposed $\text{FMA}^{\text{bf16}}_{\text{2\_2}}$ {4} or $\text{FMA}^{\text{bf16}}_{\text{1\_2}}$ 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 $\text{FMA}^{\text{bf16}}_{\text{n\_m}}$ in terms of performance speed-ups using micro-architectural simulations. We show that $\text{FMA}^{\text{bf16}}_{\text{1\_1}}$, $\text{FMA}^{\text{bf16}}_{\text{1\_2}}$, and $\text{FMA}^{\text{bf16}}_{\text{2\_2}}$ {4} operators achieve speed-ups of 1.35×, 1.34× and 1.28× on ResNet101, respectively when compared to FP32 at the equivalent area.

In future work, we plan to exploit the different $\text{FMA}^{\text{bf16}}_{\text{n\_m}}$ 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 $\text{FMA}^{\text{bf16}}_{\text{n\_m}}$ operators can help increase precision while using the cheapest FMA operator ($\text{FMA}^{\text{bf16}}_{\text{1\_1}}$) for a large portion of the computations.

Chapter 8

Conclusions

During this thesis's development, we performed a series of contributions that helped to have a reliable tool to emulate reduced precision formats and works on several DNN frameworks without additional effort. Our tool enables the evaluation and verification of different use cases.

8.1 FASE: A Fast, Accurate and Seamless Emulator

FASE enables the emulation of any numerical formats and works using DNN training frameworks like Tensorflow, PyTorch, or Caffe. Its main strength is to be easy to use, without changes on top of the original DNN training code. FASE uses Intel PIN and currently supports MP, BF16, Dynamic, and $\mathrm{FMA}^{\mathrm{bf16}}_{\mathrm{n\_m}}$ operators. It could be used as a starting point for research about reduced precision approaches to train DNN.

8.2 Use cases

We studied three use cases of FASE and tested them on ResNets, AlexNet, and Inception to solve object classification tasks using ImageNet and CIFAR datasets. GAN, RNN, and Transformer were also used to test the versatility of FASE of different DNN types.

- The first use case helps to evaluate the MP method to train 3DGAN to simulate High Energy Physics Detectors.
- The second use case is a dynamic training approach that automatically changes between reduced numerical strategies to reduce power consumption and execution time while training state-of-the-art DNNs.

- The last use case proposes a new set of $\mathrm{FMA}^{\mathrm{bf16}}_{\mathrm{n\_m}}$ operators relying completely on BF16 computation to train DNN, enabling fully compliant BF16 hardware without FP32 support.

8.3 Future Work

With the development of FASE, we can study different reduced precision approaches to train DNNs. We provide new researchers in this area with a tool that can be used to test custom numerical datatypes before their implementation in real hardware, reducing costs and increasing their reliability. These approaches can be added to the currently supported strategies in FASE. The new family of incoming DNN models is open to be done as future work; the GPT family of models is an example of them.

Additionally, FASE works on Intel hardware up to AVX512 instruction set extension. The new family of Intel processors is not supported (Sapphire Rapids); enabling FASE to use it and intercept AMX instructions is currently in process and will be added.

Finally, the implementation of FASE to support GPU, other than Intel hardware, should be included in consequent FASE releases. In this last case, we would need any Binary Analysis Tool that works as PIN but on GPU hardware [95].

References

[1] Abadi, M., Agarwal, A., and Barham, e. a. (2016). TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems.

[2] ARM (2021). Sve instructions.

[3] Athey, S. (2018). *The Impact of Machine Learning on Economics*, pages 507–547. University of Chicago Press.

[4] Audhkhasi, K., Osoba, O., and Kosko, B. (2013). Noise benefits in backpropagation and deep bidirectional pre-training. In *IJCNN*.

[5] Azzolini, V., Bugelskis, D., Hreus, T., Maeshima, K., Fernandez, M. J., Norkus, A., Fraser, P. J., Rovere, M., Schneider, M. A., et al. (2019). The data quality monitoring software for the cms experiment at the lhc: past, present and future. In *EPJ Web of Conferences*, volume 214, page 02003. EDP Sciences.

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

[7] Ben, T. (2018). Pytorch seq2seq.

[8] Brown, T. B., Mann, B., Ryder, N., and et al. (2020a). Language models are few-shot learners.

[9] Brown, T. B., Mann, B., Ryder, N., and et al. (2020b). Language models are few-shot learners.

[10] Carlson, T. E., Heirman, W., and Eeckhout, L. (2011). 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)*, pages 52:1–52:12.

[11] CERN (2023). Worldwide lhc computing grid. https://wlcg.web.cern.ch/.

[12] Chatelain, Y., Petit, E., de Oliveira Castro, P., Lartigue, G., and Defour, D. (2019). Automatic exploration of reduced floating-point representations in iterative methods. In *Euro-Par*, Lecture Notes in Computer Science. Springer.

[13] Chekalina, V., Orlova, E., Ratnikov, F., Ulyanov, D., Ustyuzhanin, A., and Zakharov, E. (2019). Generative Models for Fast Calorimeter Simulation: the LHCb case. *EPJ Web Conf.*, 214:02034.

[14] Chollet, F. et al. (2015). Keras. https://github.com/fchollet/keras.

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

[16] Courbet, C. (2021). Nsan: A floating-point numerical sanitizer. In *Proceedings of the 30th ACM SIGPLAN International Conference on Compiler Construction*, CC 2021, page 83–93, New York, NY, USA. Association for Computing Machinery.

[17] Dawson, A. and Düben, P. D. (2017). rpe v5: an emulator for reduced floating-point precision in large numerical simulations. *Geoscientific Model Development*, (6).

[18] Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., and Fei-Fei, L. (2009). ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR*.

[19] Denis, C., Castro, P. D. O., and Petit, E. (2015). Verificarlo: checking floating point accuracy through monte carlo arithmetic.

[20] Di Sipio, R., Giannelli, M. F., Haghighat, S. K., and Palazzo, S. (2020). DijetGAN: A Generative-Adversarial Network Approach for the Simulation of QCD Dijet Events at the LHC. *JHEP*, 08:110.

[21] Doersch, C. (2016). Tutorial on variational autoencoders.

[22] Durmus, A. U. (2019). Replication of recurrent neural network regularization by zaremba.

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

[24] Fei-Fei, L. (2015). Tiny imagenet visual recognition challenge. https://www.kaggle.com/c/tiny-imagenet.

[25] Fu, Y., Guo, H., Li, M., and et al. (2021). {CPT}: Efficient deep neural network training via cyclic precision. In *ICLR*.

[26] Fu, Y., You, H., Zhao, Y., and et al. (2020). Fractrain: Fractionally squeezing bit savings both temporally and spatially for efficient dnn training. In *Neurips*.

[27] Giagu, S. et al. (2020). Fast and resource-efficient deep neural network on fpga for the phase-ii level-0 muon barrel trigger of the atlas experiment. Technical report, ATL-COM-DAQ-2020-001.

[28] Golling, T., Duehrssen, M., Raine, J., Stewart, G., and Salamani, D. (2020). Fast calorimeter simulation in atlas with dnns. Technical report, ATL-COM-SOFT-2020-002.

[29] Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., and Bengio, Y. (2014). Generative adversarial nets. In Ghahramani, Z., Welling, M., Cortes, C., Lawrence, N. D., and Weinberger, K. Q., editors, *Advances in Neural Information Processing Systems 27*, pages 2672–2680. Curran Associates, Inc.

[30] Gordic, A. (2020). Original pytorch transformer model.

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

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

[33] Harper, F. M. and Konstan, J. A. (2015). The movielens datasets: History and context. *ACM Trans. Interact. Intell. Syst.*, 5(4).

[34] He, K., Zhang, X., Ren, S., and Sun, J. (2015). Deep Residual Learning for Image Recognition. pages 1–12.

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

[36] Higham, N. J. (1993). The accuracy of floating point summation. *SIAM*.

[37] Huang, G., Liu, Z., van der Maaten, L., and Weinberger, K. Q. (2018). Densely connected convolutional networks.

[38] Intel. Intel caffe framework optimization.

[39] Intel. Intel deep neural network library.

[40] Intel. Oneapi dnn library.

[41] Intel (2020a). Intel architecture instruction set extensions and future features programming reference.

[42] Intel (2020b). Intel architecture instruction set extensions programming reference.

[43] Intel (2020c). Intel math kernel library.

[44] Intel (2020d). Pin - a binary instrumentation tool.

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

[46] Jia, Y., Shelhamer, E., Donahue, J., Karayev, S., Long, J., Girshick, R., Guadarrama, S., and Darrell, T. (2014). Caffe: Convolutional architecture for fast feature embedding.

[47] Jiang, W., Higuera, J. C. G., Angles, B., Sun, W., Javan, M., and Yi, K. M. (2020). Optimizing through learned errors for accurate sports field registration. In *The IEEE Winter Conference on Applications of Computer Vision (WACV)*.

[48] Johnson, J. (2018). Rethinking floating point for deep learning.

[49] Jones, D. T. (2019). Setting the standards for machine learning in biology. *Nature Reviews Molecular Cell Biology*, 20(11):659–660.

[50] Jouppi, N. P., Young, C., Patil, N., Patterson, D., Agrawal, G., Bajwa, R., Bates, S., Bhatia, S., Boden, N., Borchers, A., Boyle, R., Cantin, P.-l., Chao, C., Clark, C., Coriell, J., Daley, M., Dau, M., Dean, J., Gelb, B., Ghaemmaghami, T. V., Gottipati, R., Gulland, W., Hagmann, R., Ho, C. R., Hogberg, D., Hu, J., Hundt, R., Hurt, D., Ibarz, J., Jaffey, A., Jaworski, A., Kaplan, A., Khaitan, H., Killebrew, D., Koch, A., Kumar, N., Lacy, S., Laudon, J., Law, J., Le, D., Leary, C., Liu, Z., Lucke, K., Lundin, A., MacKean, G., Maggiore, A., Mahony, M., Miller, K., Nagarajan, R., Narayanaswami, R., Ni, R., Nix, K., Norrie, T., Omernick, M., Penukonda, N., Phelps, A., Ross, J., Ross, M., Salek, A., Samadiani, E., Severn, C., Sizikov, G., Snelham, M., Souter, J., Steinberg, D., Swing, A., Tan, M., Thorson, G., Tian, B., Toma, H., Tuttle, E., Vasudevan, V., Walter, R., Wang, W., Wilcox, E., and Yoon, D. H. (2017). In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, page 1–12, New York, NY, USA. Association for Computing Machinery.

[51] Kalamkar, D., Mudigere, D., Mellempudi, N., Das, D., Banerjee, K., Avancha, S., Vooturi, D. T., Jammalamadaka, N., Huang, J., Yuen, H., Yang, J., Park, J., Heinecke, A., Georganas, E., Srinivasan, S., Kundu, A., Smelyanskiy, M., Kaul, B., and Dubey, P. (2019). A Study of BFLOAT16 for Deep Learning Training. pages 1–10.

[52] Khattak, G., Vallecorsa, S., and Carminati, F. (2018). Three dimensional energy parametrized generative adversarial networks for electromagnetic shower simulation. In *2018 25th IEEE International Conference on Image Processing (ICIP)*, pages 3913–3917.

[53] Khattak, G. R., Vallecorsa, S., Carminati, F., and Khan, G. M. (2019). Particle detector simulation using generative adversarial networks with domain related constraints. In *ICMLA*.

[54] Krizhevsky, A. (2009). Learning Multiple Layers of Features from Tiny Images. Technical report.

[55] Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In Pereira, F., Burges, C. J. C., Bottou, L., and Weinberger, K. Q., editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc.

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

[57] LeCun, Y., Bengio, Y., and Hinton, G. (2015). Deep learning. *Nature*, 521(7553):436–444.

[58] Lecun, Y., Bottou, L., Bengio, Y., and Haffner, P. (1998). Gradient-based learning applied to document recognition. In *Proceedings of the IEEE*, pages 2278–2324.

[59] Loroch, D. M., Wehn, N., Pfreundt, F.-J., and Keuper, J. (2017). Tensorquant - a simulation toolbox for deep neural network quantization.

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

[61] Luk, C. K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V. J., and Hazelwood, K. (2005b). Pin: Building customized program analysis tools with dynamic instrumentation. *ACM SIGPLAN Notices*, 40(6):190–200.

[62] Mater, A. C. and Coote, M. L. (2019). Deep learning in chemistry. *Journal of Chemical Information and Modeling*, 59(6):2545–2559.

[63] Metahexane (2020). Neural graph collaborative filtering algorithm in pytorch.

[64] Micikevicius, P., Narang, S., Alben, J., Diamos, G., Elsen, E., Garcia, D., Ginsburg, B., Houston, M., Kuchaiev, O., Venkatesh, G., and Wu, H. (2017). Mixed Precision Training. pages 1–12.

[65] Molchanov, P., Tyree, S., Karras, T., Aila, T., and Kautz, J. (2017). Pruning convolutional neural networks for resource efficient inference.

[66] Nvidia (2020a). Nvidia a100 tensor core gpu architecture.

[67] Nvidia (2020b). Nvidia ampere whitepaper.

[68] Nvidia (2021). Improve tensor core operations.

[69] nVidia (2022). nvidia tensor cores.

[70] OpenAI (2023). Gpt-4 technical report.

[71] Osorio, J., Armejach, A., Khattak, G., Petit, E., Vallecorsa, S., and Casas, M. (2020). Evaluating mixed-precision arithmetic for 3d generative adversarial networks to simulate high energy physics detectors. In *2020 19th IEEE International Conference on Machine Learning and Applications (ICMLA)*, pages 49–56.

[72] Osorio, J., Armejach, A., Petit, E., Henry, G., and Casas, M. (2021). Dynamically adapting floating-point precision to accelerate deep neural network training. In *2021 20th IEEE International Conference on Machine Learning and Applications (ICMLA)*, pages 980–987.

[73] Osorio, J., Armejach, A., Petit, E., Henry, G., and Casas, M. (2022a). A bf16 fma is all you need for dnn training. *IEEE Transactions on Emerging Topics in Computing*, 10(3):1302–1314.

[74] Osorio, J., Armejach, A., Petit, E., Henry, G., and Casas, M. (2022b). Fase: A fast, accurate and seamless emulator for custom numerical formats. In *2022 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 144–146.

[75] Paganini, M., de Oliveira, L., and Nachman, B. (2018). Calogan: Simulating 3d high energy particle showers in multilayer electromagnetic calorimeters with generative adversarial networks. *Phys. Rev. D*, 97:014021.

[76] Paszke, A., Gross, S., Chintala, S., and Chanan, G. (2020). Pytorch.

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

[78] Ren, J., Wu, L., and Yang, J. M. (2020). Unveiling cp property of top-higgs coupling with graph neural networks at the lhc. *Physics Letters B*, 802:135198.

[79] Rodriguez, A., Ziv, B., Fomenko, E., Meiri, E., and Shen, H. (2018). Lower numerical precision deep learning inference and training.

[80] Salamani, D. et al. (2018). Deep Generative Models for Fast Shower Simulation in ATLAS. In *Proceedings, 14th International Conference on e-Science: Amsterdam, Netherlands, October 29-November 1, 2018*, page 348.

[81] Sandler, M., Howard, A., Zhu, M., Zhmoginov, A., and Chen, L.-C. (2019). Mobilenetv2: Inverted residuals and linear bottlenecks.

[82] Simonyan, K. and Zisserman, A. (2015). Very Deep Convolutional Networks for Large-Scale Image Recognition. *ICLR Conference Paper*, pages 1–9.

[83] Stephens, N. (2019). Bfloat16 processing for neural networks on armv8-a.

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

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

[86] Sung, W. and Hwang, K. (2014). Fixed-point feedforward deep neural network design using weights +1, 0, and -1. *SiPS*, pages 174–179.

[87] Sze, V., Chen, Y. H., Yang, T. J., and Emer, J. S. (2017). Efficient Processing of Deep Neural Networks: A Tutorial and Survey. *Proceedings of the IEEE*, 105(12):2295–2329.

[88] Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., and Rabinovich, A. (2015). Going deeper with convolutions. In *Computer Vision and Pattern Recognition (CVPR)*.

[89] Tambe, T., Yang, E.-Y., Wan, Z., Deng, Y., Reddi, V. J., Rush, A. M., Brooks, D., and Wei, G.-Y. (2019). Adaptivfloat: A floating-point based data type for resilient deep learning inference.

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

[91] Tong, J. Y. F., Nagle, D., and Rutenbar, R. A. (2000). Reducing power by optimizing the necessary precision/range of floating-point arithmetic. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 8(3):273–286.

[92] Toon, N. (2020). Introducing 2nd generation ipu systems for ai at scale.

[93] Trevett, B. (2020). Neural machine translation by jointly learning to align and translate.

[94] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. (2017). Attention is all you need.

[95] Villa, O., Stephenson, M., Nellans, D., and Keckler, S. W. (2019). Nvbit: A dynamic binary instrumentation framework for nvidia gpus. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, page 372–383, New York, NY, USA. Association for Computing Machinery.

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

[97] Wang, S. and Kanwar, P. (2019). Bfloat16: The secret to high performance on cloud tpus.

[98] Wang, X., He, X., Wang, M., Feng, F., and Chua, T.-S. (2019). Neural graph collaborative filtering. In *Proceedings of the 42nd International ACM SIGIR Conference on Research and Development in Information Retrieval*. ACM.

[99] Wu, S., Li, G., Chen, F., and Shi, L. (2018). Training and Inference with Integers in Deep Neural Networks. pages 1–14.

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

[101] Yang, Y., Wu, S., Deng, L., Yan, T., Xie, Y., and Li, G. (2019b). Training high-performance and large-scale deep neural networks with full 8-bit integers.

[102] Zaremba, W., Sutskever, I., and Vinyals, O. (2015). Rnn regularization.

[103] Zeiler, M. D. and Fergus, R. (2014). Visualizing and understanding convolutional networks. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 8689 LNCS(PART 1):818–833.

[104] Zhang, T., Lin, Z., Yang, G., and Sa, C. D. (2019). Qpytorch: A low-precision arithmetic simulation framework.

[105] Zhou, S., Wu, Y., Ni, Z., Zhou, X., Wen, H., and Zou, Y. (2018). Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients.