ApproxTrain: Fast Simulation of Approximate Multipliers for DNN Training and Inference

Jing Gong\textsuperscript{\textcopyright}, Hassaan Saadat\textsuperscript{\textcopyright}, Hasindu Gamaarachchi\textsuperscript{\textcopyright}, Haris Javaid, Xiaobo Sharon Hu\textsuperscript{\textcopyright}, Fellow, IEEE, and Sri Parameswaran, Senior Member, IEEE

Abstract—Edge training of deep neural networks (DNNs) is a desirable goal for continuous learning; however, it is hindered by the enormous computational power required by training. Hardware approximate multipliers have shown their effectiveness in gaining resource efficiency in DNN inference accelerators; however, training with approximate multipliers is largely unexplored. To build resource-efficient accelerators with approximate multipliers supporting DNN training, a thorough evaluation of training convergence and accuracy for different DNN architectures and different approximate multipliers is needed. This article presents ApproxTrain, an open-source framework that allows fast evaluation of DNN training and inference using simulated approximate multipliers. ApproxTrain is as user-friendly as TensorFlow (TF) and requires only a high-level description of a DNN architecture along with C/C++ functional models of the approximate multiplier. We improve the speed of the simulation at the multiplier level by using a novel LUT-based approximate floating-point (FP) multiplier simulator on GPU (AMSim). Additionally, a novel flow is presented to seamlessly convert C/C++ functional models of approximate FP multipliers into AMSim. ApproxTrain leverages CUDA and efficiently integrates AMSim into the TensorFlow library to overcome the absence of native hardware approximate multiplier in commercial GPUs. We use ApproxTrain to evaluate the convergence and accuracy performance of DNN training with approximate multipliers for three application domains: image classification, object detection, and neural machine translation. The evaluations demonstrate similar convergence behavior and negligible change in test accuracy compared to FP32 and Bfloat16 multipliers. Compared to CPU-based approximate multiplier simulations in training and inference, the GPU-accelerated ApproxTrain is more than 2500\times faster. Based on highly optimized closed-source cuDNN/cuBLAS libraries with native hardware multipliers, the original TensorFlow is, on average, only 8\times faster than ApproxTrain.

Index Terms—Approximate multiplier, approximate TensorFlow (TF), deep neural network (DNN) training.

I. INTRODUCTION

THE TRAINING phase in deep learning is significantly more computationally demanding than the inference phase. Recent works have shown the importance of moving training to the edge to perform continuous learning [1], [2], though such deployments are scarce due to the high training cost. Thus, training has so far been largely relegated to high-performance computers and the cloud. To support training at the edge, besides the accuracy of the learning model, energy/power and area efficiency are paramount.

An efficient DNN system could be realized through two distinct yet complementary approaches: 1) by exploring and finding efficient DNN hardware implementation schemes for both training and inference and 2) by exploring and finding the most suitable DNN architecture for the given problem. In the context of hardware implementation, multipliers are one of the most compute-intensive hardware elements within a deep learning system [3]. Thus, using approximate multipliers is a promising scheme for efficient implementation of DNN systems [4], and the efficacy of various approximate multiplier designs in DNN inference in terms of model accuracy and implementation efficiency has been demonstrated in recent years [5]. However, DNN training using approximate multipliers is largely unexplored. This article presents ApproxTrain,\textsuperscript{1} a framework that allows fast evaluation of DNN training when using a variety of simulated approximate multipliers.

A major obstacle in the exploration and evaluation of approximate multipliers in DNN training of different architectures is the absence of native hardware support for customized approximate floating-point (FP) multipliers on commercial processing units (CPUs, GPUs, and NPUs). Hence, a software simulation of approximate multiplications (AFMs) is needed for handling DNN training with approximate multipliers. ApproxTrain facilitates thorough evaluations of training of many different DNNs with differing approximate multipliers in a user-friendly manner with practically feasible runtime to find suitable approximate multipliers that can be integrated into edge devices for continuous learning.

\textsuperscript{1}https://github.com/AaronJing/ApproxTrain

1937-4151 © 2023 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.
Thorough and effective evaluation of approximate multiplier designs in DNN training has two practical requirements: 1) the runtime of the training simulation should be sufficiently fast and practically feasible and 2) the ability to describe DNN architectures at a high level so that various DNN architectures can be quickly constructed and evaluated. However, resorting to software simulation makes it challenging to meet the above requirements. First, since training is time-consuming, inefficient simulation may further slow down training (up to orders of magnitude), making its runtime practically infeasible.

Second, the standard frameworks, such as TensorFlow (TF) and Pytorch, which allow a high-level description of DNN architectures and take advantage of the computational power of commercial GPUs/TPUs/NPUs are not equipped to utilize approximate multipliers. This is because these frameworks invoke the native multipliers (built in the hardware) of these commercial platforms. Moreover, these deep learning frameworks, although open-source themselves, are based on closed-source cuDNN and cuBLAS libraries [6] at their backend. Hence, any modification in these high-level frameworks for incorporating fast approximate multiplier simulation while preserving their flexibility, requires specialized parallel programming skills, making it a daunting task for a regular DNN developer or approximate multiplier designer.

ApproxTrain overcomes the above challenges through a three-level approach. First, to improve the speed of ApproxTrain at the multiplier level, we use a mantissa lookup table (LUT)-based approach for functional simulation of approximate FP multipliers. Our mantissa-lookup approach is based on the observation that only mantissa is approximated in most state-of-the-art approximate multipliers [7], [8]. Therefore, we propose to compute the mantissa using a LUT, whereas to compute the sign and exponent using the standard approach. This enables smaller LUTs as opposed to constructing LUTs for the full bit-width of the operands. The proposed mantissa LUT-based approach makes the speed performance independent of the type of approximate multipliers (i.e., the speed performance becomes consistent across different approximate multipliers).

Any training mechanism based on CPU can be painfully slower than the standard GPU implementations. Therefore, our second contribution, aiming to improve system-level performance, is the development of GPU-accelerated custom CUDA kernels as an alternative for nonmodifiable closed-source cuDNN and cuBLAS libraries. Our GPU-accelerated custom CUDA kernels exploit various architectural features of the GPU (such as fine-grained and coarse-grained parallelism, memory coalescing, and on-chip shared memory).

Third, to preserve the flexibility and ease of use of the high-level framework (TensorFlow), we create custom approximate TF ops (explained in Section III) and seamlessly integrate them into TensorFlow. These custom TF ops support different types of DNN layers with approximate multiplication. The GPU-accelerated custom CUDA kernels (described above) are used to implement our custom approximate TF ops.

ApproxTrain requires only standard TensorFlow-based implementation of DNN architectures, along with C/C++-based functional models of the approximate multipliers. Overall, ApproxTrain is a DNN framework that: 1) allows fast DNN training evaluations with approximate multipliers; 2) is similar and as user-friendly as the popular high-level frameworks such as TensorFlow; and 3) simplifies the daunting task of integrating approximate multiplier simulation in TensorFlow (or similar frameworks) to make it transparent to the DNN architecture/multiplier designer. The key contributions of this article are summarized below.

1) A novel workflow is presented to seamlessly transform C/C++ functional models of any approximate FP multipliers provided by designers into a LUT-based simulator (AMSim). This LUT-based AMSim requires negligible GPU memory compared to a whole LUT-based method that stores all multiplication results into LUT.

2) We present ApproxTrain, a novel framework built on the TensorFlow platform for efficient evaluation of deep neural network (DNN) training using simulated approximate FP multipliers. The framework replaces all multiplications in both forward and backpropagation, thereby facilitating training and inference with approximate multipliers. Notably, ApproxTrain offers the flexibility of allowing users to specify different multipliers for each layer by simply loading a different LUT during layer instantiation in the TensorFlow model.

3) Since the closed-source cuDNN and cuBLAS libraries cannot be modified to integrate functional models of approximate multipliers into ApproxTrain, we developed GPU-accelerated custom CUDA kernels.

4) To explore the prospects of DNN training with approximate multipliers, we use ApproxTrain to evaluate DNN training with approximate multipliers in terms of training convergence and accuracy results. Our experiments with small and large datasets (including ImageNet) using popular neural network architectures demonstrate successful convergence with negligible change in test accuracy.

5) The ApproxTrain framework, as well as the developed custom CUDA libraries, are released as open-source [9]. There has been an attempt for such a framework. However, that framework only supports DNN inference with 8-bit approximate integer multipliers [5]. Thus, to our best knowledge, there is no general framework that supports fast and user-friendly DNN training as well as inference with approximate FP multipliers (i.e., in both forward and backpropagation phases of training).

A. Paper Organization

Section II motivates the usage of approximate multipliers in training. The background and related work are discussed in Section III and Section IV, respectively. Section V describes the novel LUT-based approximate FP multiplier simulation, and Section VI describes the presented framework ApproxTrain. Experimental evaluation setup using ApproxTrain is elaborated in Section VII. The training convergence and accuracy results using approximate multipliers are presented in Section VIII. The evaluation of the runtime performance of ApproxTrain is presented in Section IX. Finally, Section X concludes this article.
approximate multipliers, which is the focus of this article. Situates the fast evaluation framework for DNN training using results for different DNN architectures is needed. This necessitates a thorough evaluation of training convergence and accuracy when using FP32, FP16, or Bfloat16 formats. However, before hardware multipliers to enable more resource-efficient training than observed that the approximate multipliers are more power-efficient and more area-efficient than the FP32, FP16, and Bfloat16 multipliers. Hence, we expect the approximate hardware multipliers to be used in multilayer perceptrons (MLPs) as feedforward layers. The core of the dense layer is matrix–vector multiplication. Convolution and depthwise convolution are widely used in image classification tasks, such as ResNet and MobileNets in Table I. Convolution consists of multiplication-intensive operations. Pooling layers are responsible for down-sampling, reducing the feature maps size, and not involving multiplications. The Multihead Attention layer has shown extraordinary performance in language and image tasks, and the famous example is Transformer, as shown in Table I. The MultiheadAttention layer involves matrix multiplication under the hood.

**DNN Training:** Training is an iterative process that finds optimal parameters (weights) to reduce the difference between the model prediction and the dataset labels. Initially, the weights are randomly generated based on a particular distribution. Then, a series of forward propagation and backpropagation phases are executed until no further accuracy improvement occurs. Training is highly resource hungry and processes a large number of multiplications.

The mixed-precision training procedure [11] is shown as a flow diagram in Fig. 3. In mixed-precision training, accumulations are performed in single precision (FP32), whereas the multiplications may be performed with smaller bit-width datatypes such as FP16 or Bfloat16. This technique is the de facto standard that has been used in many DNN hardware platforms, such as Google TPU [12] and NVIDIA GPU [15]. In Fig. 3, the conversion from FP32 to the smaller datatypes is referred to as quantization. The quantization step is omitted when the multiplications are also performed in FP32 format.

**III. BACKGROUND**

**A. Deep Neural Networks**

**DNN Structure:** A DNN consists of an input layer, multiple hidden layers, and an output layer, as shown in Fig. 2. Each layer is composed of multiple parallel neurons; for instance, the hidden layer in Fig. 2 has four parallel neurons. Each parallel neuron contributes to a weighted output that acts as an input for the next layer. Inputs and weights are subjected to linear algebra operations in each layer, followed by nonlinear activation functions. Table I shows common types of layers used in some common DNN architectures. The dense layer is used by all architectures listed since it can be used as a classification layer at the output of most DNN architectures. It can be simply used in multilayer perceptrons (MLPs) as feedforward layers. The core of the dense layer is matrix–vector multiplication. Convolution and depthwise convolution are widely used in image classification tasks, such as ResNet and MobileNets in Table I. Convolution consists of multiplication-intensive operations. Pooling layers are responsible for down-sampling, reducing the feature maps size, and not involving multiplications. The Multihead Attention layer has shown extraordinary performance in language and image tasks, and the famous example is Transformer, as shown in Table I. The MultiHeadAttention layer involves matrix multiplication under the hood.

**B. Background**

In computer number representation, the dynamic range of a number format refers to the range between the largest and the smallest (nonzero) number that can be represented.
Fig. 3. DNN training algorithm with two stages: forward propagation and backpropagation.

Fig. 3 depicts that the activations from layer $l-1$ are multiplied with quantized weights in the current layer $l$, and results are accumulated and quantized to become the activation for the next layer. After forward propagation, an error is calculated to reflect the difference between the predicted result and the label, known as propagation error. Propagation error is propagated backward to calculate gradients for weights in each layer so that the DNN model can achieve better prediction. As shown in Fig. 3, Errors$^{l+1}$ is propagated backward to layer $l$. To get WeightsGradient$^{l}$, multiplyaccumulation and quantization are performed on Errors$^{l+1}$ and Activations$^{l-1}$ in succession. Then Weights$^{l}$ could be updated with the calculated gradients. Similarly, Errors$^{l}$ (preceding layer gradient) is computed as the quantized multiply-accumulation between the Weights$^{l}$ and the Errors$^{l+1}$ backpropagated from the succeeding layer $l+1$.

TensorFlow (TF): TensorFlow, one of the most popular deep learning libraries, has been used to develop many DNNs across different application domains. TensorFlow provides highly abstracted $Ops$ such as Conv2D (2-D Convolution, also known as Conv2D layer) and Dense (also known as fully connected layer) that are commonly used across different architectures and applications, allowing the users to build models easily. In TensorFlow, DNN architectures are represented as graphs, and the $Ops$ are nodes that take one or more tensors (multidimensional arrays) as inputs and perform computations on those tensors. Every $Op$ has its Compute method that defines the mathematical operation to be performed on the tensors. Each $Op$ typically has a corresponding gradient $Op$, which is used in the backpropagation in the training phase. The backend of TensorFlow utilizes cuBLAS and cuDNN libraries developed by NVIDIA for GPU acceleration. Both closed-source libraries are highly optimized low-level primitives for linear algebra and DNN.

B. Approximate Computing and Approximate Multipliers

Approximate hardware for DNNs typically refers to using approximate arithmetic units for DNNs inference. In the computation of DNNs, multiplications dominate operations and consume the most power and area. Thus, utilizing approximate multipliers could improve inference efficiency and has been extensively studied [8], [18]. For example, Saadat et al. [8] replaced accurate multipliers with minimally biased multipliers in AlexNet during the inference stage. Other works [18], [19] all show significant energy savings when using approximate multipliers with minimal quality degradation. These works use only a limited variety of DNNs, support only inference, and do not report the runtime of simulations. TFApprox [5] enables fast and flexible simulation of approximate multipliers in DNN inference due to TensorFlow integration and GPU acceleration. However, TFapprox is limited to 8-bit integer multiplications and supports inference only.

Most of these efficiency gains were made for DNN inference. Few works focus on training DNNs with approximate multipliers. Kim et al. [18] claimed approximate multipliers are not suitable for training because DaDianNao [20] project failed to train the DNN with the fixed-point datatype. Fixed-point 16-bit has a limited dynamic range that prevents some necessary small gradients from being represented. Hammad et al. [21] attempted to train a DNN with approximate multipliers first and further improved convergence with accurate multipliers. However, only VGGnet was evaluated; thus, its feasibility is not shown on a wide variety of neural networks. The work in [22] first attempted to train neural networks with logarithm-based approximate multipliers. However, the evaluated model is simple: only a fully connected layer was considered. A fully connected layer is well known to be redundant; for example, Han et al. [23] achieved a compression rate of 40 times for LeNet-300-100 (MLP), with parameters reducing from 1070 to 27 kb. Therefore, the convergence could have been well caused by redundancy. Given these concerns, it is challenging to show the efficacy of the work in [22].

To overcome the above limitations, we first present a novel LUT-based approximate FP multiplier simulator on GPU (AMSim) that could efficiently simulate any type of approximate FP multipliers. Then, we integrate the AMSim into the ApproxTrain framework, supporting training and inference.

V. AM SIMULATION: EFFICIENT LOOKUP TABLE-BASED APPROXIMATE FP MULTIPLIER SIMULATION

Evaluation of approximate multipliers in DNN training requires all multiplications in the forward and backpropagation (previously depicted in Fig. 3) to be replaced by AFMs. Moreover, since there is no native hardware support for approximate FP multipliers on CPUs, a software
simulation of approximate FP multipliers is needed. This is shown in Fig. 4 where all the multiplications are replaced with AMSim (approximate multiplier simulation). The requirement of efficiently simulating approximate FP multiplier in software with low-performance overhead (in order to overcome the problem of nonexistent native hardware support for approximate multipliers on CPUs) poses a few challenges. First, different approximate FP multipliers have different computation procedures (approximation schemes), so direct C/C++ simulation (bit manipulation) cannot guarantee consistent low overhead independent of the type of approximate FP multiplier. Second, approximate multiplier designers may find it challenging to optimize multiplier simulation speed. Furthermore, GPU-based simulation must be utilized to efficiently couple approximate FP multipliers with training and inference algorithms. Therefore, this section presents a novel flow (AMSim) for seamlessly converting the C/C++ simulation implementation into an optimal LUT-based approximate FP simulator on GPU.

LUT generation (see Section V-A) takes user-defined multiplier C/C++ code as input and generates mantissa products LUT, as depicted in the red dash box in Fig. 5. This generation step is required to be run only once for a given approximate FP multiplier, and the generated LUTs are written into binary files; thus, multipliers designers could load LUT binary files during runtime. Upon completing this generation step, users may load these LUTs into AMSim (see Section V-B) during runtime. This flow is designed based on the following key observations: 1) mantissa multiplications contribute to 91.10% area and 92.70% power in the circuit of accurate FP multiplications [24]; thus, most AM designs [7], [8], [24] target optimizing the mantissa multiplications stage, and keep existing computation of exponent and sign unchanged and 2) different designs have differing approximate mantissa multiplication procedures, making it challenging to develop an efficient approximate FP multiplier simulators that will fit all designs. Mantissa multiplications are, therefore, simulated by LUTs (see LUT Generation below), and the whole procedure for approximate FP multiplications involves three steps (see AMSim below): 1) retrieve mantissa multiplication results from LUT; 2) compute sign and exponent; and 3) concatenate sign, exponent, and mantissa multiplication to achieve the final approximate multiplication result.

Algorithm 1: Approximate Mantissa Multiplications LUT Generation

```c
Algorithm 1 Approximate Mantissa Multiplications LUT Generation

1: function APPROXIMATE MANTISSA MULTIPLICATIONS LUT GENERATION
2: A ← empty FP32; B ← empty FP32
3: Sign(A) ← 0 or 1; Sign(B) ← 0 or 1
4: Exponent(A) ← N; Exponent(B) ← K; ∀N, K ∈ [1, 254]; ∀(N + K − 127) ∈ [1, 254]
5: for k = 0 to 2^M do
6: for j = 0 to 2^M do
7: Mantissa(A) ← k; Mantissa(B) ← j
8: C ← approx_mul(A, B)
9: un_normalized_exp ← Exp(A)+Exp(B)-127
10: Carry ← 0
11: if un_normalized_exp < Exponent(C) then
12: Carry ← 1
13: end if
14: mantmul_lut[k × 2^M + j] ← (Carry × 2^M + Mantissa(C))
15: end for
16: end for
17: end function
```

A. Lookup Table Generation

Vaverka et al. [5] stored all approximate integer multiplication results in LUT, with the LUT occupying only 128 kB of GPU memory. However, this solution is not practical for approximate FP multipliers. The de-facto industrial standard for FP is 16 bits (BFloat16 and FP16), and storing the entire product LUT would require 8.6 GB of memory, which is too costly for GPU. We propose to store only the mantissa multiplication results in LUTs based on our above-mentioned observations. In the case of BFloat16, there are 7 mantissa bits, resulting in 2^7 × 2^7 (stored as 1 byte in LUT) = 16 KB, which is negligible compared to 8 GB memory in GTX1080.

Algorithm 1 takes the bit-width of mantissa M and approximate FP multiplication C/C++ code `approx_mul` to generate mantissa multiplication LUTs. In lines 2–4 of Algorithm 1, two FP numbers A and B are initialized with arbitrary signs and exponents since the mantissa product is independent of signs and exponents. It should be noted that the exponent of A and B and the exponent of their product must not be special cases (0, Inf, and NaN). Otherwise, the carry from the mantissa multiplication cannot be detected; the detailed conditions are presented in line 4. In our AMSim, the carry from the mantissa multiplication is used to adjust the exponent. Lines 5–16 of Algorithm 1 capture the nested loop used to generate all possible mantissa combinations. The mantissa of A and B are populated by the nested loop indices in line 7. The populated A and B are then passed into the user-defined C/C++ function, `approx_mul`. Then, `approx_mul` generates an approximate FP product C. Lines 9–13 of Algorithm 1 describe how to
Algorithm 2 AMSim

```
input: a, b, mntmultlut = a and b are FP inputs to the simulation. mntmultlut is the mantissa product lookup table
output: c = c is the approximate product of a and b

1: global variables
2: M, Mantissa Bit-width.
3: M_MASK, Mantissa Mask.
4: E_MASK, Exponent Mask.
5: end global variables
6: function APPROXIMATE FP MULTIPLICATION SIMULATION
7: A0 = M_MASK & a, B0 = M_MASK & b
8: Mntmult = mntmultlut(A0) (23 - M x 2) +
9: B0 = (23 - M)
10: if Exp ≤ 0 or a & E_MASK == 0 or b & E_MASK == 0 then
11: c ← 0
12: else if Exp ≥ 255 then
13: c ← INFINITY
14: else
15: c ← Sign (Exp) Mntmult
16: end if
17: return c
18: end function
```

detect carry without knowing any details about how hardware or simulations are implemented. The unnormalized exponent (un_normalized_exp) of C is calculated in line 9 and is compared with the real exponent return by the user-defined C/C++ function (approx_mul) in lines 11–13 to set carry. In AMSim, the carry bit needs to be retained in order to adjust the exponent if the real exponent of C is greater than the unnormalized exponent of C. Finally, the carry bit and the mantissa results are stored in the same entry of LUT (mntmultlut) in line 14. A script has been provided for multiplier designers to generate LUTs on the condition that a C/C++ AFM function is properly implemented by the designer.

B. AMSim

The AMSim is proposed to simulate approximate FP multipliers on GPU; it is composed of the three steps mentioned earlier in Section V, and Algorithm 2 elaborates this mechanism in detail. Algorithm 2 takes two FP numbers a, b, and the mantissa product LUT; whereas it outputs the approximate product of a and b. The quantization shown in Figs. 2 and 4 is implicitly handled by M_MASK in line 2 of Algorithm 2, since this quantization only requires truncation of mantissa part. In line 7, the mantissa of A and B are extracted; then, on line 8, the index to fetch LUT is computed by concatenating the mantissa of A and B. In lines 9 and 10 of Algorithm 2, the mantissa multiplication results and carry are decoupled. In line 10, the sign of the approximate multiplication output C is computed as the XOR (exclusive-or) of the signs of A and B. Exceptional cases (0 and Infinity) and normal cases are handled from lines 11 to 17 of Algorithm 2. If either the biased exponent of C is not greater than zero or one of the inputs (A or B is zero) (line 12), then C should be zero. When the biased exponent of C exceeds or equals 255, the C is overflowed and results in Infinity. In lines 16–18 of Algorithm 2, the biased exponent is adjusted based on carry in the normal case. As a final step, sign, exponent, and mantissa are concatenated to form C.

The AMSim is implemented as an inline device function and compiled into a part of the CUDA kernel. The LUT is retrieved from the GPU global memory by texture cache on GPU, a similar approach to that described in work [5]. Note that, despite giving 16-bit FP as an example, our approach enables generic \((1, e, m)\) FP approximate multiplication simulation; bits of mantissa \(m\) could be selected from 1 to 10, supporting a wide range of precisions. Additionally, the bits of the exponent \(e\) can be varied from 1 to 8, provided that a proper exponent casting function is given.

C. Memory Overhead of AMSim

The approximate mantissa product LUT is stored in GPU global memory and is fetched into the computational kernel through the texture cache; thus, the DNN workload in the L1 cache is not affected. Texture cache is separate from L1 cache/shared memory. The LUT memory overheads of different mantissa bit-width are listed in Table II. These memory overheads that range only from 1KB to 1MB are negligible compared to 32GB global memory in NVIDIA V100 or 8GB in NVIDIA GTX1080. Specifically, given mantissa bit-width \(m\), total entries are computed as \(2^m \times 2^m\). Each entry consists of a mantissa product with bit-width \(m\), and one carry bit from the normalization step (refer to Algorithm 1). Thus, the number of bits for each entry is \(m+1\). For approximate multipliers with mantissa bit-width less than 8, the mantissa product and carry bit can be fitted into uint8_t (1 byte); otherwise, the mantissa product and the carry bit must be fitted into uint16_t (2 bytes). We can observe that even when the mantissa bit-width is 10, memory overhead is 1MB; it occupies only 0.01% of GTX1080 DRAM, which is negligible.

D. GEMM Performance Evaluation

To demonstrate the efficacy of the proposed AMSim, we compare the timing performance of AMSim with the direct C simulation for different approximate multipliers in GEMM (general matrix multiplication). The two input matrices to GEMM are both 8000 by 8000, and the experiment is performed using GTX1080 GPU. The results are depicted in Fig. 6, which show that our approach, AMSim, is consistently 2× slower than native hardware FP32 for REALM16, AFM16, and MIT16. On the other hand, the direct C simulations have higher as well as variable performance overhead (between 4.6× and 78.2×) for the different multipliers.
VI. APPROXTRAIN

ApproxTrain integrates AMSim into TensorFlow so that different DNN architectures with different approximate multipliers can be efficiently constructed and evaluated using high-level APIs. In ApproxTrain, we create custom TF ops (see Section III for the explanation of ops) to support different types of DNN layers with different approximate multipliers. In these custom TF ops of ApproxTrain, all multiplications (both in forward and backpropagation) are replaced by approximate multiplications, as depicted in Fig. 4. To equip our custom ops with AMSim, we developed GPU-accelerated custom CUDA kernels for the implementation of our custom TF ops. These custom CUDA kernels are needed because the standard ops available in the open-source TF library use closed-source cuDNN/cuBLAS libraries in the backend that cannot be modified. Thus, ApproxTrain enables fast evaluation of training/inference of different DNN architectures using different approximate multiplier designs.

In the following subsections, we first present an overview of ApproxTrain, followed by a description of our approach to create four custom TF ops and the underlying CUDA kernels.

A. ApproxTrain: Framework Overview

An overview of the ApproxTrain from the user’s perspective is shown in Fig. 7. In addition to the normal design flow of TF, a user simply needs to: 1) provide functional models of the approximate multipliers in C/C++ and 2) replace the standard layer ops with the approximate versions from ApproxTrain in the DNN architecture. Example code snippets for such a replacement is demonstrated in Listing 1 and Listing 2. After importing the compiled library of the custom ops from ApproxTrain, the DENSE op (fully connected) and CONV2D op (convolutional layer) are simply replaced with their approximate versions AMDENSE and AMCONV2D, respectively.

The overview of the internals of creation and compilation of our custom ops in ApproxTrain is depicted in Fig. 5. The main component is the approximate operator C++ class inside the blue dashed box, which has multiple operations such as input validation, serializing tensors to linear arrays, memory allocation, and performing computations. The computational part of the approximate operator C++ class includes functions to calculate feedforward propagation and backpropagation by invoking our custom CUDA kernels or CPU kernels.1 Custom CUDA kernels (Section VI-D) are responsible for linear algebra operations and data rearrangement and are equipped to use AMSim. The AMSim is implemented as a device function for running on GPU. As stated before, custom CUDA kernels are written from scratch because the closed-source cuDNN and cuBLAS libraries cannot be modified to use approximate multipliers.

All CUDA kernels are compiled by NVCC, and the C++ operator class is compiled with g++. Then, the compiled C++ object files are linked with the compiled CUDA kernel objects to form the approximate operator-shared library. This approximate operator runtime library is then enclosed in a python wrapper which is then registered into the standard TF library. Note that the compilation steps above only need to be done once. Instead of replacing the corresponding original operators in TensorFlow, the new approximate operators are kept alongside the original ones. Given user-defined approximate multiplier C codes, LUTs can be obtained by LUT Generation (explained in Section V-A), as depicted in Fig. 5. The obtained LUTs are loaded into the approximate operator runtime library during runtime to simulate different functional models of approximate multipliers. These python wrappers have the same parameters as the original operators, and the users simply need to change the name of the original operators to the approximate ones to simulate the approximate multipliers, as demonstrated in the code listings above.

Currently, our framework contains four operators that cover a large portion of DNN architectures: 1) AMCONV2D 2) AMDENSE and 3) AMCONV2D and 4) AMDENSE.

The CPU implementation was used for validating our GPU implementation and benchmark, but could also be used by a user who does not have GPU access at the cost of higher runtime.

Listing 1. DNN model using standard TensorFlow op for convolutional and dense layer.

Listing 2. DNN model using ApproxTrain for approximate convolutional and dense layers.

1The CPU implementation was used for validating our GPU implementation and benchmark, but could also be used by a user who does not have GPU access at the cost of higher runtime.
Algorithm 3: Approximate Forward Propagation

```
input: A^(l−1), W_l, S, P, LUT \rightarrow A^l: activation from layer l – 1; W_l: weight from layer l; S: stride; P: Padding; LUT: mactiss product LUT
output: A^l  \rightarrow A^l: activation from layer l

1: function APPROXIMATE FORWARD PROPAGATION
2:   PSize,ColSize ← calculate_sizes(A^(l−1),W_l,P,S) \rightarrow PSize: The size of padding ColSize: The size of Im2Col results
3:   allocate_GPU_memory(PSize,ColSize,LUT) \rightarrow For A^(l−1), Columns(output of IM2Col), W_l, S, and LUT.
4:   Columns ← IM2COL_kernel(A^(l−1), PSize, ColSize)
5:   A_l ← GEMM_kernel(Columns, W_l, LUT) \ldots \rightarrow Refer to m, n, k, lda, ldb and ldc that are omitted for simplicity. A_l is the output activation
6: end function
```

Algorithm 4: Approximate Backpropagation

```
input: A^(l−1), W_l, Error⁺⁺, Stride \rightarrow A^l is the activation from layer l – 1, W_l is the weight from layer l
output: W_l, Error⁺⁺ \rightarrow W_l is the gradient of W_l

1: function APPROXIMATE BACKPROPAGATION
2:   PSize,ColSize ← calculate_sizes(A^(l−1), W_l, Error⁺⁺) \rightarrow PSize: The size of padding ColSize: The size of Im2Col results
3:   allocate_GPU_memory(PSize,ColSize,LUT) \rightarrow For A^(l−1), Columns, DilatedError⁺⁺, W_l and Errors⁺⁺
4:   Columns_error ← IM2COL_WKernel(A^(l−1), Error⁺⁺, ColSize, PSize)
5:   W_l ← GEMM_Kernel(Columns, W_l, LUT) \ldots
6:   Columns_dilError⁺⁺ ← IM2COL_PDGKernal(Errors⁺⁺, ColSize, PSize) \rightarrow IM2COL kernel for preceding layer gradient (PLG), PDError⁺⁺ is the PaddedDilatedError⁺⁺
7:   (W_l^T) V_{l−1} ← Reverse_Transpose_kernel(W_l)
8:   Errors⁺⁺ ← GEMM_Kernel(Columns, PDError⁺⁺, (W_l^T, LUT) \ldots
9: end function
```

As opposed to forward propagation, mapping backpropagation to GEMM along with IM2COL to efficiently exploit GPU architecture is challenging. A native method to map the computation of weight gradient to GEMM would be to implement a separate GPU kernel to perform the dilation followed by the GEMM kernel. However, this separate kernel would add extra performance overhead and require several times the memory as the original array. Instead of such a native approach, we implicitly perform this dilation inside the IM2COL_WK_Kernel (a modified IM2COL kernel) by skipping elements in A^(l−1) that correspond to zero (line 4 of Algorithm 4) if the Error⁺⁺ array was dilated.

2) Preceding Layer Gradient: We also restructure the computation of the preceding layer gradient to exploit the IM2COL+GEMM approach as shown in Fig. 8(c). For this, we first subject Errors⁺⁺ to dilation (inserting zeros between elements as explained before), followed by padding (inserting zeros around the image along height and width dimensions). This PaddedDilatedErrors⁺⁺ is fed as the input to the GEMM operation as shown in Fig. 8(c) along with the output of IM2COL.

Exploiting GEMM in the preceding layer gradient [Errors⁺⁺ in Fig. 8(c)] of AMCONV2D for efficient execution on the GPU is even more nontrivial since both transposition and reversal of elements in Weights⁺⁺ are involved, in addition to padding and dilation of Errors⁺⁺. Dilation operation of Errors⁺⁺ is integrated into the IM2COL_PDGKernal (a modified IM2COL Kernel that performs padding and dilation), where each thread copies a zero into IM2COL results if the current pixel is at a dilated position; thus, it avoids overhead from invoking an additional kernel. Additionally, a separate kernel that solely performs reversal and transposition of QuantizedWeights⁺⁺ is invoked before the GEMM operation to enable memory coalescing of the GEMM operation.

Since AMCONV2D is implemented by the GEMM approach, all multiplications are done in the GEMM kernel; thus, we replace accurate multiplication in the GEMM with AMSim device function to enable simulation.
C. Other Custom Ops With Approximation

In addition to the AMCONV2D op described above, we created AMDENSE, AMMATMUL, and AMMHA custom ops to support Dense, MATMUL, and Multihead Attention layers, respectively. The operations in the dense layer can be mapped to matrix-vector multiplication. We implemented a separate matrix-vector multiplication CUDA kernel for this rather than using the previously used GEMM kernel because shared memory-based tiling is superfluous for a 1-D vector. The AMMATMUL op relies on our custom GEMM kernel. The Multihead Attention layer is composed of multiple dense layers and MATMUL layers. So we use our AMDENSE and AMMATMUL to realize AMMAHA op. Due to space limitations in the manuscript, the detailed description of the creation of these custom ops is presented in the supplementary document available at ApproxTrain GitHub repository [9].

D. Custom CUDA Kernels

The custom AM ops described above utilize several custom CUDA kernels. We developed these kernels to replace the kernels offered by the closed-source cuDNN and cuBLAS libraries. These custom CUDA kernels (which involve multiplication) are equipped to use AMSim to perform multiplication. In simple terms, these kernels may call the approximate multiplier functions with two operands as the arguments instead of using the "*" operator to multiply the two operands. The developed custom kernels include: GEMM kernel, IM2COL kernel, IM2COL_Weight kernel, IM2COL.PLG kernel, Transpose-And-Reverse kernel, and Matrix-Vector Multiplication kernel. Due to space limitations in the manuscript, the detailed description of these custom kernels is presented in the supplementary document available at ApproxTrain GitHub repository [9].

VII. EXPERIMENTAL SETUP

We use ApproxTrain to perform a series of training and inference experiments for three types of ML tasks: image classification, object detection, and neural machine translation. The purpose of these experimental evaluations is twofold: 1) evaluate the efficacy of approximate multipliers in DNN training for both forward propagation and backpropagation, i.e., the effect on training convergence and accuracy (Section VIII) and 2) evaluate the timing performance of ApproxTrain with different DNN architectures/datasets/platforms (Section IX).

In the experiments, the framework inputs are neural network architecture, dataset, and the type of multiplier. The outputs are the timing performance numbers and accuracy performance of deep learning tasks. The experiments use various DNN architectures and datasets on different platforms. The details of different datasets, neural network architectures, and other settings used in our experiments are below.

Datasets: For the image classification task, three popular datasets are used in our experiments: MNIST [28], CIFAR-10 [29], and ImageNet [30]. MNIST is hand-written digits consisting of 60,000 training and 10,000 test samples. Each sample is a 32×32 gray image. In the CIFAR10 dataset, there are ten classes with 6000 32×32 colored images per class. The dataset is divided into 50,000 training samples and 10,000 test samples. ImageNet contains 1.2 million training images spanning 1000 object classes. MNIST and CIFAR-10 are usually considered small datasets, while ImageNet is one of the largest datasets available for image classification.

For the object detection task, PASCAL VOC 2007 [31] is used to the train object detection model (R-CNN). This dataset consists of 5k training/validation images and 5k test images over 20 objects. For the neural machine translation task, the Portuguese-English translation dataset from TED Talks Open Translation Project is used [32]. This translation dataset contains 52,000 training, 1200 validation, and 1800 test examples.

Neural Network Architectures: Five neural network architectures: LeNet-300-100, LeNet-5 [33], ResNet-18/34/50 [34] are used in image classification experiments. LeNet-300-100 is an MLP, while LeNet-5 is a convolutional neural network (CNN) having two convolution layers and three dense layers. ResNet is a deep convolutional network whose complexity can be adjusted by adding or removing building blocks [34]. ResNet-18/34/50 contains 18, 34, and 50 layers, respectively. Faster R-CNN is used to evaluate the object detection task. Faster R-CNN mainly contains three modules to propose object bounding, convolutional network, regional proposal network, and classifier. For neural machine translation, the Transformer model [35] is used which contains four transformer blocks. The various combinations of datasets and architectures used in our experiments are listed in the first and second columns of Table IV.

Datatype: Our experiments use FP format instead of integer/fixed-point because training typically requires a higher dynamic range. We keep the sign (s) as 1-bit and the exponent (e) as 8-bit (similar to FP32 and Bfloat16 [12]). The number of mantissa bits (m) is varied in different experiments to achieve different bit-width. The details of various datatypes/multipliers and their bit-width are listed in Table III. Since exponents are the same in all formats, type conversion is simply a matter of bit-truncation or bit-extension. All accumulation operations are performed in FP32 to realize the industry de-facto standard of mixed-precision training when lower bit-width is used for multiplication [15].

Experiment Platforms: Three types of platforms are used for the experiments. System-1 is equipped with a single NVIDIA
TABLE III
DATATYPES AND MULTIPLIERS USED IN EXPERIMENTS

| Multiplier/ Datatype | Bit-width | Description |
|----------------------|-----------|-------------|
| FP32 (1,8,23)        |           | IEEE 754 standard format |
| BFpfloat16 (1,8,7)   |           | Brain Floating Point format [12] |
| AFM32 (1,8,23)       |           | 32-bit version of approx. mult AFM [8] |
| AFM16 (1,8,7)        |           | 16-bit version of approx. mult AFM [8] |

V100 GPU and 12-core Intel Xeon Scalable (Cascade Lake) processor (24 CPUs per core). System-II is equipped with GTX1080 and i7 6600 CPU. System-I and System-II are used for runtime performance benchmarking (Section IX). In addition, to run the training convergence test for large datasets (Section VIII), we used another system which is a two-node cluster equipped with eight V100 GPUs and two full Intel Xeon Scalable “Cascade Lake” cores. To realize multi-GPU (distributed) training environment on the high-end cluster, TensorFlow wrapped by Horovod is used. The operating system used on all systems is Ubuntu 18.04.

Implementation Details: The presented framework is integrated into TensorFlow 2. The tested TensorFlow version is 2.3.0, which requires CUDA 10.1 and cuDNN 7.6.5. The custom CUDA kernels are compiled with NVCC provided by CUDA 10.1, whereas the supporting C/C++ files are compiled with gcc-8/g++-8.

VIII. RESULTS: TRAINING ACCURACY EVALUATION

In this section, we present the evaluation of training accuracy and convergence using the approximate multipliers [8] with ApproxTrain. For the following experiments, AFM32 and AFM16 [8] are used as representative approximate multipliers, whereas FP32 and BFpfloat16 formats are used as the baseline. Fig. 1 depicted area and power efficiency of AFM16 and AFM32. In comparison with the FP32 multiplier, the AFM32 is 12× smaller and 24× more energy efficient, while the AFM16 is about 20× smaller and 50× more energy efficient. The different combinations of dataset/NN-architectures used in the experiments are listed as the title of each graph in Fig. 9 (and also listed in the first two columns of Table IV). For example, in Fig. 9(a), the MNIST dataset is used with LeNet-300-100 architecture.

A. Training Convergence and Test Results

The training convergence results are depicted in Fig. 9, where the training accuracy (y-axis) is plotted against train-epochs (x-axis) for the four multipliers listed in Table III. The weights and parameters for the NN architectures are randomly initialized; however, for a given NN/dataset combination, the same random seed is used for all four multipliers (for fair comparison among different multipliers). The training is run for several epochs until the validation accuracy stabilizes. The training converges in 20 or fewer epochs for the two LeNets, whereas it stabilizes in around 100 epochs for CIFAR-10/ResNet combinations. From Fig. 9, we observe that the training-accuracy plots for the AFM32 and AFM16 closely follow the plots for FP32 and BFpfloat16. The observation applies to all of the evaluated datasets/NN-architecture combinations. In other words, training converges with approximate multipliers (AFM32 and AFM16), and the convergence behavior and convergence rate are the same as for FP32 and BFpfloat16. Note that, as shown in Fig. 1, AFM32 and AFM16 are much smaller and more power-efficient than FP32 and BFpfloat16 multipliers.

The final test accuracy results for the eight dataset/architecture combinations are reported in Table IV for 32 and 16-bit formats. The third and sixth columns, presenting results for FP32 and BFpfloat16, are considered baselines for 32 and 16-bit formats, respectively. The difference between test accuracy for AFMs compared to the corresponding baselines is listed in the fifth and eighth columns. From Table IV, for both data formats, we observe that the test accuracy for all dataset/architecture combinations using approximate...
multipliers is very similar to the baseline (accuracy degradation is within 0.10% for image classification, 0.0% for object detection, and 0.35% for neural machine translation). Note that similar accuracy differences also exist between the heavily adapted FP32 and Bfloat16 formats (columns 3 and 6–Table IV). Therefore, we argue that such degradation is acceptable. In fact, in most cases, the accuracy for approximate multipliers is slightly better than the baselines (highlighted in blue in the table). A reason for this is that the error injected by erroneous AFMs in training can be considered stochastic noise, which is a type of regularization.

B. Cross-Format Test Accuracy

For the ImageNet dataset, we perform another experiment where we evaluate the test accuracy with a multiplier that is different from the one used for training. In other words, we train the neural network using one multiplier type and test it using another. The purpose of this experiment is to observe if any drastic over-fitting occurs w.r.t. the used multiplier type.

The results of the experiment are listed in Table V. The multipliers used for training are listed along the second column, while the multipliers listed across the second row are used for testing. Essentially, the numbers in the diagonal (highlighted in bold) are the test results when the same multiplier is used for training and testing and are considered the baseline for each row. The rest of the entries in Table V depict test-accuracy results when different multipliers are used for testing. We observe that the difference in accuracy is within 0.10%, which we deem acceptable, as discussed in the previous section. Therefore, this experiment demonstrates that we may safely train and deploy a neural network with different multiplier types (including approximate multipliers) as they do not drastically over-fit for the given data/multiplier type.

C. Pruning With Approximate Multiplier

We also performed an experiment to couple pruning with the use of approximate multipliers in training. Pruning is a technique for the efficient inference that requires repeated training. Thus, it is beneficial to improve training efficiency and demonstrate that our framework enables hardware/algorithm co-design. The pruning code/algorithm is implemented following the official TensorFlow example. First, a CNN with two convolution layers and three dense layers was pretrained for 20 epochs. The pretrained CNN weights are loaded into a new model to be pruned. After pruning, the model is retrained for two epochs to refine accuracy. The horizontal dashed–dotted line in Fig. 10 represents the baseline for all the other experiments. Orange, blue, and purple curves indicate FP32, Bfloat16, and AFM16 test accuracy against sparsity, respectively. They declined slowly from 70% to 80% sparsity and rapidly afterward. We observed all three curves are above the baseline from 70% to 80%. In pruned Bfloat16 and pruned AFM16, 83% is the optimal sparsity level. However, FP32 dropped below the baseline with 83% sparsity. As a result, AFM16 could replace Bfloat16 as a drop-in replacement since its curves are consistently above the baseline. By coupling approximate multipliers design with a pruning algorithm, we demonstrate our framework’s flexibility.

IX. RESULTS: RUNTIME PERFORMANCE EVALUATION

As discussed in Section I, the aim of ApproxTrain is to perform DNN training with approximate multiplier simulation in practically feasible runtime. In this section, we present results for a detailed evaluation of the timing performance of ApproxTrain for training as well as for inference.

The overall timing performance comparison is evaluated by recording the average time for DNNs to train/infer one batch. The results are listed in Tables VI and VII. For these evaluations, the training and inference experiments are run on two platforms: System-I and System-II (described in Section VII). For both training and inference, and for both platforms, Tables VI and VII present four types of runtime measurements. These are:

1) **TFnG**—runtime for training/inference performed using standard TensorFlow with cuDNN/cuBLAS libraries on GPU with native hardware multiplier (FP32);
TABLE VI
TRAINING RUNTIME RESULTS ON SYSTEM-I AND SYSTEM-II

| Dataset | Neural Network | System - I (V100 GPU) | System - II (GTX1080 GPU) |
|---------|----------------|------------------------|---------------------------|
|         |                | Actual Time per batch  | Actual Time per batch     |
|         |                | TF with native multi.  | TF with native multi.      |
|         |                | AT with native mut.    | AT with A/M               |
|         |                | AT with native mut.    | AT with A/M               |
|         |                | ATxG/TFxG (slower)     | ATxG/TFxG (slower)        |
| MNIST   | LeNet-10       | 2.0 ms, 3 ms, 3 ms    | 0.9 ms, 2 ms, 3 ms, 3 s  |
| MNIST   | LeNet-5        | 3 ms, 7 ms, 13 ms, 23 | 2.3 ms, 9 ms, 16 ms, 22 s|
| CIFAR10 | ResNet18       | 13 ms, 49 ms, 178 ms | 27 ms, 120 ms, 374 ms, 672s|
| CIFAR10 | ResNet34       | 23 ms, 90 ms, 338 ms, 1376ms | 40.0 ms, 15.0 ms, 4072 ms|
| CIFAR10 | ResNet50       | 44 ms, 154 ms, 478 ms, 1632 ms | 3.5 ms, 10.8 ms, 34177 ms|
| ImageNet| ResNet50       | 114 ms, 460 ms, 1646 ms, 4896 ms | 4.0 ms, 12.8 ms, 3343 ms|

TABLE VII
INFERENCE RUNTIME RESULTS ON SYSTEM-I AND SYSTEM-II

| Dataset | Neural Network | System - I (V100 GPU) | System - II (GTX1080 GPU) |
|---------|----------------|------------------------|---------------------------|
|         |                | Actual Time per batch  | Actual Time per batch     |
|         |                | TF with native multi.  | TF with native multi.      |
|         |                | AT with native mut.    | AT with A/M               |
|         |                | AT with native mut.    | AT with A/M               |
|         |                | ATxG/TFxG (slower)     | ATxG/TFxG (slower)        |
| MNIST   | LeNet-10       | 1 ms, 1 ms, 2 ms, 1 s  | 0.085 ms, 1 ms, 1 ms, 657 ms|
| MNIST   | LeNet-5        | 2 ms, 3 ms, 4 ms, 8 s  | 1.7 ms, 3 ms, 4 ms, 7 s   |
| CIFAR10 | ResNet18       | 5 ms, 15 ms, 56 ms, 320 s | 3.0 ms, 11.3 ms, 5745 ms|
| CIFAR10 | ResNet34       | 9 ms, 25 ms, 107 ms, 576 s | 2.9 ms, 12.2, 5405 ms    |
| CIFAR10 | ResNet50       | 14 ms, 36 ms, 131 ms, 544 s | 2.5 ms, 9.2 ms, 4154 ms  |
| ImageNet| ResNet50       | 15 ms, 110 ms, 398 ms, 1580 s | 3.2 ms, 11.4 ms, 4993 ms |

2) \(ATnG\)—runtime for training/inference performed using ApproxTrain with custom CUDA kernels (described in Section VI-D) on GPU with native hardware multiplier (FP32);

3) \(ATxG\)—runtime for training/inference performed using ApproxTrain with custom CUDA kernels on GPU with AMSim (16-bit FP datatype (1, 8, 7) in Table III); and,

4) \(ATxC\)—runtime for training/inference performed using ApproxTrain with custom CUDA kernels on CPU with direct C/C++ simulation of approx. multiplier.

The \(TFnG\) values, i.e., the runtime of standard TensorFlow with native hardware multiplication supported by GPU, are considered baseline in the following discussion. Note that we did not perform runtime evaluation experiments with Bfloat16 since the available hardware did not natively support Bfloat16. Meanwhile, 16-bit FP datatype (1, 8, 7) as shown in Table III is used in AMSim, considering it equivalent to the industry de-facto standard for training/inference.

1) Custom CUDA Kernels in ApproxTrain Versus Optimized cuDDN/cuBLAS in TensorFlow: For this comparison, we use ApproxTrain with the “\(*\)” operator for multiplication, which invokes the native hardware multiplier on GPU instead of an approximate multiplier simulation model. This comparison demonstrates the performance of custom kernels on GPUs with native multiplier hardware. Therefore, in columns 4 and 11 of Tables VI and VII, \(ATnG\) refers to the time with our custom CUDA kernels in ApproxTrain, as opposed to columns 3 and 10 which refers to the time taken by cuDDN/cuBLAS-based TensorFlow.

The slow-down (speed-ratio) of \(ATnG\) (ApproxTrain with a native multiplier on GPU) compared with \(TFnG\) (TF with a native multiplier on GPU) is highlighted in bold (black) in Tables VI and VII. In the training phase, ApproxTrain with native multiplication is \(1.5 \times -5 \times\) slower than standard TensorFlow for the various datasets. This less than \(5 \times\) slowdown is reasonable because the standard TensorFlow is based on highly optimized but closed-source CuDNN and CuBLAS libraries. These closed-source CuDNN and cuBLAS libraries have been optimized by teams of several hundred professionals within Nvidia for over a decade. Due to their closed-source nature, they cannot be modified to incorporate approximate multiplier simulation. Therefore, to realize ApproxTrain, we developed our own GPU-accelerated custom CUDA kernels as alternates to the required kernels from the CuDNN and CuBLAS libraries. Since our framework and custom CUDA kernels are open-source, the research community may contribute to further optimizations.

2) ApproxTrain Perf. With Approx. Multiplier Simulation: We compare the runtime of ApproxTrain with approximate multiplier simulation on GPU \(ATxG\) against the standard TF with native multiplication on GPU \(TFnG\). The slow-down (speed-ratio) of this comparison is highlighted in bold (blue) in Tables VI and VII. Slow-down is around 2 \(\times\) for the smallest dataset/architecture, whereas the slow-down for ImageNet is about 13 \(\times\). This comparison demonstrates the performance penalty of approximate multiplication simulation plus the use of a custom CUDA library. Essentially, the difference in bold-black and bold-blue slow-downs is due to the overheads of the approximate multiplier simulation. The slow-down numbers for System-II are slightly lower than for System-I for training and inference in general since V100 GPU has Tensor Cores, which are faster than the architecture in the GTX1080.

Previous work TFapprox with inference-only framework [5] has shown 10 \(\times\) slowdown for small dataset/architectures, despite only supporting 8-bit integer datatype. ApproxTrain supports FP training and inference with just 7.32 \(\times\) slowdown (7.32 \(\times\) is the geometric mean of all experiments).
containing both AMDENSE and AMCONV2D operators). Despite 16-bit FP datatype (Bfloat16) being used to benchmark ApproxTrain, we compare inference performance on approximate CONV2D operators of ApproxTrain and TFappox. We reproduce the TFappox project [36] and benchmark both ApproxTrain and TFappox with identical measurement procedures in System-II (GTX1080 GPU). As shown in Fig. 11, similar inference performance can be observed for both ApproxTrain and TFappox across the four different datasets/architectures, containing intensive approximate CONV2D operations. Note that TFappox only supports 8-bit integer inference for approximate CONV2D operator while ApproxTrain enables generic (1, e, m) FP training and inference at once for both AMCONV2D and AMDENSE operators.

3) ApproxTrain GPU Performance Versus CPU-Based Approximate Multiplier Simulations: We compare the performance of ApproxTrain with GPU (ATxG) with the runtime of approximate multiplier training/inference on CPU (ATxC). The speed-ups of these comparisons are highlighted in bold (green) in Tables VI and VII. We observe that for training on System-I, the geometric mean speed-up is more than 2500x! Similarly, for inference, the speed-up is more than 2869x. The speed-ups for System-II are slightly lower since the GPU is less powerful, while CPUs in System-II have similar performance to System-I. Thus, the presented ApproxTrain offers a fast and easy solution for testing approximate multipliers and DNNs compared to naive simulations on the CPU.

X. Conclusion
This article proposed a framework (ApproxTrain) to perform training and inference with approximate FP multipliers through simulation. First, a novel flow is proposed to effortlessly convert C/C++-based functional models of the approximate multipliers into optimal AMSim. Then, this AMSim is integrated into ApproxTrain (an extension of Tensorflow), leveraging CUDA to speed up the simulation. ApproxTrain allows researchers to flexibly evaluate and explore their approximate multiplier designs in various DNNs. Our evaluations show that approximate multipliers (AFM) could converge DNNs as well as FP32 and Bfloat16 multipliers. The GPU runtime shows a significant speedup over CPU runtime, making it practically feasible. ApproxTrain is released as open-source [9] for further contributions from the research community.

Acknowledgment
This research/project was undertaken with the assistance of resources and services from the National Computational Infrastructure (NCI), which is supported by the Australian Government. The authors acknowledge Yikai Wang for monitoring experiments.

References
[1] M. Aledhari, R. Razzaq, R. M. Parizi, and F. Saeed, “Federated learning: A survey on enabling technologies, protocols, and applications,” IEEE Access, vol. 8, pp. 140699–140725, 2020.
[2] H. Cai, C. Gan, L. Zhu, and S. Han, “TinyTL: Reduce memory, not parameters for efficient on-device learning,” in Proc. Adv. Neural Inf. Process. Syst., 2020, pp. 11285–11297. [Online]. Available: https://proceedings.neurips.cc/paper/2020/file/8117acab4d11274cf65c200d7ed56b4-Paper.pdf
[3] S. Jain, S. Venkataaramani, V. Srivivasan, J. Choi, P. Chuang, and L. Chang, “Compensated-DNN: Energy efficient low-precision deep neural networks by compensating Quantization errors,” in Proc. 55th ACM/ESDA/EU Design Autom. Conf. (DAC), 2018, pp. 1–6.
[4] G. Zervakis, H. Saadat, H. Amrouch, A. Gerstlauer, S. Parameswaran, and J. Henkel, “Approximate computing for ML: State-of-the-art, challenges and visions,” in Proc. 26th Asia South Pacific Design Autom. Conf. (ASP-DAC), 2021, pp. 189–196.
[5] F. Vavkerka, V. Mrazek, Z. Vasicke, L. Sekanina, M. A. Hanif, and M. Shafique, "TFApprox: Toward simulation of DNN approximate hardware accelerators on GPU," in Proc. Design Autom. Test Europe Conf. (DATE), 2020, pp. 294–297.
[6] "Deep learning frameworks." 2019. Accessed: Apr. 7, 2022. [Online]. Available: https://developer.nvidia.com/deep-learning-frameworks/view
[7] H. Saadat, H. Javaid, A. Ignjatovic, and S. Parameswaran, “REALM: Reduced-error approximate log-based integer multiplier,” in Proc. Design Autom. Test Europe Conf. Exhibit. (DATE), Mar. 2020, pp. 1366–1371.
[8] H. Saadat, H. Bokhari, and S. Parameswaran, “Minimally biased multipliers for approximate integer and floating-point multiplication,” IEEE Trans. Comput.-Aided Design Integr. Circuits Syst., vol. 37, no. 11, pp. 2623–2635, Nov. 2018.
[9] J. Gong, H. Gamaarachchi, and H. Saadat, “ApproxTrain,” 2022. [Online]. Available: https://github.com/AaronJing/ApproxTrain
[10] B. Chniel, L. Ben-Uri, M. Shkolnik, E. Hofer, R. Banner, and D. Soudry, “Neural gradients are near-lognormal: Improved quantized and sparse training,” in Proc. Int. Conf. Learn. Represent., 2021.
[11] P. Micikevicius et al., “Mixed precision training,” in Proc. Int. Conf. Learn. Represent., 2018.
[12] S. Wang and P. Kanwar, “BFloat16: The secret to high performance on cloud TPUs,” Aug. 2019. Accessed: May 6, 2020. [Online]. Available: https://cloud.google.com/blog/products/ai-machine-learning/
[13] “NVIDIA TENSOR CORES unprecedented acceleration for HPC and AI.” Accessed: Dec. 16, 2022. [Online]. Available: https://www.nvidia.com/en-us/data-center/tensor-cores/
[14] A. V. Cueva. “Intel® deep learning boost new deep learning instruction bfloat16.” Jun. 2020. Accessed: Feb. 25, 2022. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-deep-learning-boost-new-instruction-bfloat16.html
[15] A. Vishwanath. “Video series: Mixed-precision training techniques using tensor cores for deep learning.” Jan. 2019. Accessed: Apr. 7, 2022. [Online]. Available: https://developer.nvidia.com/blog/video-mixed-precision-techniques-tensor-cores-deep-learning/
[16] S. Mittal, “A survey of techniques for approximate computing.” ACM Comput. Surveys, vol. 48, no. 4, pp. 1–33, Mar. 2016. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2891449.2893356
[17] T. Yuan, W. Liu, J. Han, and F. Lombardi, “High performance CNN accelerators based on hardware and algorithm co-optimization,” IEEE Trans. Circuits Syst. I, Reg. Papers, vol. 68, no. 1, pp. 250–263, Jan. 2021.
[18] M. S. Kim, A. A. D. Barrm, L. T. Oliveira, R. Hermida, and N. Bagherzadeh, “Efficient Mitchell’s approximate log multipliers for convolutional neural networks,” IEEE Trans. Comput., vol. 68, no. 5, pp. 660–675, May 2019.
[19] W. Liu, J. Xu, D. Wang, C. Wang, P. Montuschi, and F. Lombardi, “Design and evaluation of approximate logarithmic multipliers for low power error-tolerant applications,” IEEE Trans. Circuits Syst. I, Reg. Papers, vol. 65, no. 9, pp. 2856–2868, Sep. 2018.
[20] Y. Chen et al., “DaDianNao: A machine-learning supercomputer,” in Proc. 47th Annua. IEEE/ACM Int. Symp. Microarchitecture., 2014, pp. 609–622.
[21] I. Hammad, K. El-Sankary, and J. Gu, “Deep learning training with simulated approximate multipliers,” in Proc. IEEE Int. Conf. Robot. Biomimetics (ROBIO), 2019, pp. 47–51.
[22] Z. Niu, H. Jiang, M. S. Ansari, B. F. Cockburn, L. Liu, and J. Han, “A logarithmic floating-point multiplier for the efficient training of neural networks,” in Proc. Great Lakes Symp. VLSI, New York, NY, USA, 2021, pp. 65–70. [Online]. Available: https://doi.org/10.1145/3453688.3461509
[23] S. Han, H. Mao, and W. J. Dally, “Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding,” Proc. Int. Conf. Learn. Represent. (ICLR), 2016.
[24] H. Saadat, “Design and optimization of approximate multipliers and dividers for integer and floating-point arithmetic,” Ph.D. dissertation, School Comput. Sci. Eng., Univ. New South Wales, Sydney, NSW, Australia, 2021.
[25] J. N. Mitchell, “Computer multiplication and division using binary logarithms,” IRE Trans. Electron. Comput., vol. EC-11, no. 4, pp. 512–517, Aug. 1962.
[26] K. Chellapilla, S. Puri, and P. Simard, “High performance convolutional neural networks for document processing,” in Proc. 10th Int. Workshop Front. Handwriting Recognit., Oct. 2006. [Online]. Available: https://hal.inria.fr/inria-0012631
[27] S. Chetlur et al. “cuDNN: Efficient primitives for deep learning.” 2014. [Online]. Available: http://arxiv.org/abs/1410.0759.
[28] L. Deng, “The mnist database of handwritten digit images for machine learning research,” IEEE Signal Process. Mag., vol. 29, no. 6, pp. 141–142, Nov. 2012.
[29] A. Krizhevsky, “Learning multiple layers of features from tiny images,” 2009. [Online]. Available: https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf
[30] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, “ImageNet: A large-scale hierarchical image database,” in Proc. IEEE Conf. Comput. Vis. Pattern Recognit., 2009, pp. 248–255.
[31] M. Everingham, S. M. A. Eslami, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman, “The pascal visual object classes challenge: A retrospective,” Int. J. Comput. Vis., vol. 118, no. 1, pp. 98–136, Jan. 2015.
[32] Q. Ye, S. Devendra, F. Matthieu, P. Sarguna, and N. Graham, “When and why are pre-trained word embeddings useful for neural machine translation,” in Proc. HLT-NAACL, 2018, pp. 529–535.
[33] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, “Gradient-based learning applied to document recognition,” Proc. IEEE, vol. 86, no. 11, pp. 2278–2324, Nov. 1998.
[34] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” in Proc. IEEE Int. Conf. Comput. Vis. Pattern Recognit. (CVPR), 2016, pp. 770–778.
[35] A. Vaswani et al., “Attention is all you need,” in Proc. 31st Int. Conf. Neural Inf. Process. Syst., Red Hook, NY, USA, 2017, pp. 6000–6010.
[36] Z. Sun, V. Mazek, “TFapprox.” 2020. [Online]. Available: https://github.com/ehw-fit/TF-approx