TrainBF: High-Performance DNN Training Engine Using BFloat16 on AI Accelerators

Zhen Xie(✉), Siddhisanket Raskar, Murali Emani, and Venkatram Vishwanath

Argonne National Laboratory, Lemont, IL 60439, USA
{zhen.xie,sraskar,memani,venkat}@anl.gov

Abstract. Training deep neural networks (DNNs) with half-precision floating-point formats is widely supported on recent hardware and frameworks. However, current training approaches using half-precision formats neither obtain the optimal throughput due to the involvement of single-precision format nor achieve state-of-the-art model accuracy due to lower numerical digits. In this work, we present a new DNN training engine, named *TrainBF*, which leverages a typical half-precision format BFloat16 to maximize training throughput while ensuring sufficient model accuracy. TrainBF deploys BFloat16 across the entire training process for best throughput and improves model accuracy by introducing three proposed normalization techniques. TrainBF is also lightweight by only applying these normalization techniques to the layers that are most critical to model accuracy. Furthermore, TrainBF implements a parallel strategy that parallelizes the execution of operators in DNN training to make use of the spare memory space saved by half-precision for better throughput. Evaluating with six common DNN models and compared with the state-of-the-art mixed-precision approach, TrainBF achieves competitive model accuracy with an average throughput speedup of 1.21×, 1.74×, and 1.16× on NVIDIA A100 GPU, AMD MI100 GPU, and an emerging AI accelerator SambaNova, respectively.

1 Introduction

Recent advancements in Artificial Intelligence (AI) fueled by the resurgence of Deep Neural Networks (DNNs) have a spectacular success in widespread fields. Meanwhile, the increasingly complex DNN models require tremendous overhead for training. As a result, there has been broad interest in leveraging half-precision formats to reduce the training time [21]. A lot of DNN training frameworks support various half-precision formats to offer significant speedups [5,14,22].

Among them, `Float16` is a typical half-precision format, which consists of a sign bit, a 5-bit exponent, and a 10-bit fraction. Compared with the customized single-precision format TensorFloat-32 (TF32) that is used as the default format in NVIDIA Ampere architecture, Float16 has the same length of fraction bits, but shorter exponent bits, causing a narrower dynamic range of the representation than that of TF32. Thus, training DNN models with Float16 often

J. Cano et al. (Eds.): Euro-Par 2023, LNCS 14100, pp. 458–473, 2023.
https://doi.org/10.1007/978-3-031-39698-4_31

encounters *overflow* and *underflow* problems [15], which could degrade model accuracy or even lead to non-convergence.

To solve this problem, a training approach called *mixed-precision* training [14, 18,20,21] is proposed. However, the mixed-precision training with Float16 is far from achieving the theoretical performance improvement due to the involvement of single-precision format. Mixed-precision training introduces a master copy of the weights [21] in single-precision and a component called auto-casting [14] to avoid overflow problem. Also, a component called loss scaling [21] is presented to prevent underflow problem. These new components introduce a number of additional operations and incur considerable overhead. Experiments [16] show that, compared to TF32 training, mixed-precision training with Float16 brings an average throughput speedup of 1.34× using 12 common DNN models on an NVIDIA A100 GPU, which is lower than the theoretical performance speedup of 2×, because of the above three additional components.

Fortunately, such high overhead can be avoided by using another half-precision format, Brain Floating Point (BFloat16) [4], since it has the same length of exponent bits as TF32 and hence keeps the same dynamic range of representation. As a result, there is no overflow and underflow problems, and it becomes possible to avoid the involvement of single-precision and format conversions. In this paper, we will reintroduce BFloat16 format into DNN training. The motivation of this work is to achieve higher training throughput by applying BFloat16 format on all DNN operators. Thus, BFloat16 training, in nature, stores all the training data and model parameters, and performs all the computation operators in BFloat16 format entirely. However, current BFloat16 training cannot work well because of the following three challenges.

Accuracy Challenge. Recent studies [32] have shown that training DNN models in BFloat16 format alone can result in 17.3%–35.9% accuracy loss compared to training in single-precision format. The reason for accuracy loss is that, compared to single-precision, BFloat16 has only 7-bit fraction, which makes the stored data more inaccurate in numerical precision, resulting in the absence of partial model information. The more essential reason is that BFloat16 optimizes the overflow and underflow problems at the cost of sacrificing decimal precision, while the distribution of training data does not occupy the entire dynamic range of BFloat16, and therefore the exponent bits in BFloat16 is underutilized.

Overhead Concern. Even though there are some methods (will be described next) that can be applied to DNN layers to improve the bit utilization of BFloat16, these operations are accompanied by a certain overhead. For example, if we add such operations to each layer in DNN model to improve the floating-point bit utilization and amend model accuracy, the training throughput will be greatly affected and the performance advantage of half-precision will be lost. Thus, how to apply these operations to layers is another challenge.

Parallel Efficiency. Using BFloat16 format entirely in training will result in almost half of the memory (47.2% on average) being idle [2]. Traditional methods of improving memory usage by increasing batch size may lead to a compromise

in model accuracy and numerical instability due to extra noise and unstable loss function. Thus, to utilize more memory and bring higher throughput, parallel execution strategy must be redesigned without changing batch size.

To address these first challenges, we introduce a DNN training engine using BFloat16 format, named TrainBF; TrainBF is accuracy-aware to training data by optimizing the offset of sign bit and maximizing the variance of data distribution; TrainBF is overhead-aware to training throughput by applying normalization selectively. TrainBF is parallel-aware to execution efficiency by parallelizing training operators on multiple execution streams. We also evaluate TrainBF with six typical DNN models, including three convolutional neural networks (CNNs), a recurrent neural network (RNN), a graph neural network (GNN), and a scientific model on three AI accelerators. TrainBF consistently outperforms the state-of-the-art mixed-precision training approach and leads to an average of 1.21× (up to 1.67×), 1.74× (up to 1.83×), and 1.16× (up to 1.18×) speedup on NVIDIA A100 GPU, AMD MI100 GPU, and an emerging AI accelerator SambaNova, respectively.

2 Preliminaries

We now establish important preliminaries and discuss work related to ours.

Half-Precision Formats: Half-precision formats have gathered significant interests in the industry and academia over the past few years [5,14,21,22].

Two formats namely `Float16` and `BFloat16` are the most popular half-precision formats and are supported by Google TPUs, NVIDIA GPUs, AMD Instinct MI GPUs, and the emerging AI accelerators, such as the next-generation dataflow processor SambaNova. Compared to single-precision format (Float32), Float16 has a 5-bit exponent and a 10-bit fraction thus resulting in a narrow dynamic range (from 65504 to $2e^{-14}$) due to fewer fraction bits, and BFloat16 retains the same number of exponent bits (8-bit) as Float32 and therefore covers the same dynamic range but at a lower numerical precision (7-bit fraction).

Both two half-precision formats have higher performance than single-precision on the existing AI accelerators. For example, Float16 and BFloat16 can provide 16× the theoretical performance of single-precision and 2× the theoretical performance of TensorFloat-32 (TF32, which has an 8-bit exponent and a 10-bit fraction, and it is a new optimized implementation for single-precision format in NVIDIA Ampere architecture) on NVIDIA A100 GPU. However, when training with Float16, many studies [16,17] have shown that lots of additional components are introduced to avoid underflow and overflow problems, thus resulting in unavoidable overhead. Thus, this paper selects BFloat16 as the basic half-precision format in DNN training engine to avoid such overhead.

Various Training Data in DNN Training: There are three kinds of training data involved in DNN training, namely, activations, weights, and gradients. Concretely, the intermediate result in CNN models, the hidden state in RNN models, and the activation matrix in GNN models are regarded as *activations*.

The weights in CNN models, the weights of the hidden state in RNN models, and the weight matrix in GNN models are considered as *weights*. The gradients of all the weights in DNN models is regarded as *gradients*. The computation between the three kinds of training data is the main numerical computation in DNN training. In addition, the distribution of these training data is not the same [7], therefore, we will give specific optimization techniques to improve the bit utilization of each training data in BFloat16.

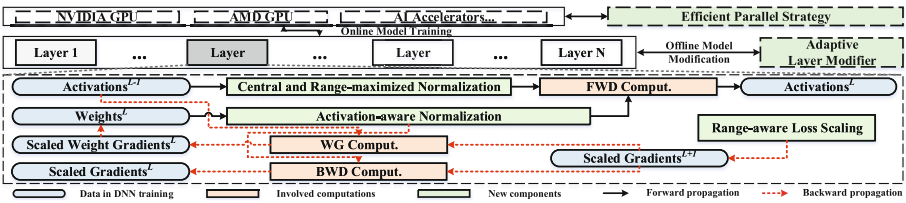

Fig. 1. Overview of TrainBF.

Essential Reasons for Accuracy Loss with Bfloat16: Besides, as per the floating-point computation theory, when adding or multiplying numbers with very different exponents can introduce a significant *floating-point error problem* [8,15]. For example, if we add $1.2 * 2^{45}$ and $3.4 * 2^{-5}$ in Float32 will yields the result of $1.2 * 2^{45}$, which drops the small one. Such error is even more pronounced when the distribution of these data is completely different and short fraction of Bfloat16 is used.

More seriously, the floating-point error caused by using low-precision format in the first few layers of DNN models will propagate to subsequent layers along with training proceeds, resulting in *error amplification problem*. The amplified computation error in the last layer can distort the main numerical information and greatly affect model accuracy.

Therefore, how to amend the information loss when converting from single-precision to Bfloat16 format, alleviate the floating-point error in computations, and avoid the error amplification problem will be the main focus in this paper.

3 Overview of TrainBF

We propose a high-performance DNN training engine using BFloat16 on AI Accelerators, called TrainBF. Figure 1 outlines its main components. TrainBF improves the training accuracy of DNN models in BFloat16 format by proposing *three normalization techniques* to optimize the data distribution of three kinds of training data. In addition, TrainBF introduces a lightweight module, *adaptive layer modifier*, to apply these normalization techniques with minimal overhead while ensuring model accuracy. Furthermore, TrainBF parallelizes the execution of training operators using an *efficient parallel strategy* on AI accelerators.

The workflow of TrainBF is divided into offline and online parts. Offline part starts from selecting the appropriate layers to apply normalization through adaptive layer modification (Sect. 5). In each selected layer, the activations is normalized to construct a bits utilization-friendly data distribution (Sect. 4.1). TrainBF normalizes its weights using the same mean and variance of the normalized activations (Sect. 4.2). During backward propagation, the training loss is amplified by a loss scaling factor provided by range-aware loss scaling to construct scaled gradients (Sect. 4.3). Next, the scaled gradients of weights is descaled and the weights is updated. In addition, online part analyses the data dependencies between operators and execute them in parallel with multiple streams under the management of its runtime component (Sect. 6).

4 Normalization Techniques in TrainBF

In this section, we will introduce three techniques to solve the problems of low bits utilization and inconsistent data distribution between different training data.

4.1 Central and Range-Maximized Normalization for Activations

As an important training data, activations are involved in all computations in forward and backward propagation to compute the gradients of previous layer and the weights. If the bits utilization of activations can be improved and the data distribution of weights and gradients can be shifted closer to it accordingly, the accuracy of numerical computations can be greatly improved, thereby amending model accuracy.

Based on our observations and existing work, the data distribution of activations is random and not centralized. Hence, the decentralized distribution cannot make full use of the sign bit in BFloat16 format due to unequal numbers of positive and negative values [1,15]. The most extreme case is when all the data is positive or negative, the sign bit is meaningless for storage. In addition, the activations are not evenly distributed across all numerical ranges in BFloat16, which makes it impossible to make full use of the exponential bits, thus resulting in very low bits utilization. For example, if all values are distributed from 2^k to 2^{k+1} in an extreme case, then the exponent bits are also meaningless.

Therefore, we propose a central and range-maximized normalization (CR_Norm) for activations, which is used to build a normalized data with zero-mean distribution and makes its values are evenly distributed in a wider data range to maximize the *number of exponent ranges* used by activations. We can apply CR_Norm after activations are generated, or replace the existing batch normalization layer [13], which is widely applied in almost all DNN models to ensure that the data is standardized over each mini-batch.

Maximizing the number of exponent ranges used by activations can improve the utilization of exponent bits, however, the disadvantage of training with such data is that it will lead to gradient explosion and oversensitive to input problems due to excessive variance. Therefore, CR_Norm designs a learnable parameter

R_{max} and includes R_{max} to loss function to trade-off between the maximum variance of normalized data and model accuracy.

The workflow of CR_Norm is shown in Algorithm 1. Algorithm 1 takes the activations over a mini-batch as input. Algorithm 1 includes two predetermined parameters ϕ and η to adjust the weight in loss function and the learning rate of R_{max}, respectively. In Algorithm 1, A represents the values of activations over a mini-batch, μ_A and σ^2 are the mean and variance of A. ϵ is the minimal amount (negligible) introduced to prevent division by zero. O is the output of CR_Norm and its variance is controlled by the learnable parameter R_{max}. L is the original loss function. In forward pass, the memorized statistics, including mean and variance of activations in m mini-batches is calculated (Line $5-6$). Then, the normalization of A has two steps: step 1 standardizes the activations A to a new distribution $\hat{A}$ with zero-mean and unit-variance (Line 7); step 2 scales $\hat{A}$ to a new distribution O with zero-mean and a new variance of the learnable parameter R_{max} (Line 8). In backward pass, R_{max} is added to loss function with a predetermined learning rate ϕ (Line 10). Then, the gradients are calculated (Line 11) and R_{max} is updated with a predetermined learning rate η (Line 12).

Algorithm 1. Normalization for Activation

1: Input: Values of activation over a mini-batch ($A = A^1, A^2, ..., A^m$)
2: Input: Parameter to be learned: R_{max}. Predetermined parameters: ϕ in loss function and η in R_{max} update
3: Output: $O^i \leftarrow CR\_Norm(A^i)$
4: **Forward Propagation:**
5: $\mu_A \leftarrow \frac{1}{m}\sum_{i=1}^{m} A^i$ //memorized mean
6: $\sigma^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m}(A^i - \mu_A)^2$ //memorized variance
7: $\hat{A}^i \leftarrow \frac{A^i - \mu_A}{\sqrt{\sigma^2+\epsilon}}$ //step 1: standardization
8: $O^i \leftarrow R_{max} * \hat{A}^i$ //step 2: scaling function
9: **Backward Propagation:**
10: Loss with range-maximized: $L = L - \phi R_{max}$
11: Compute Gradients: $\frac{\partial \ell}{\partial O}$, $\frac{\partial O}{\partial R_{max}}$, and $\frac{\partial \ell}{\partial R_{max}}$
12: Update Parameter: $R_{max} := R_{max} - \eta \frac{\partial \ell}{\partial R_{max}}$

4.2 Activation-Aware Normalization for Weights

In the process of forward propagation, a large amount of computation occurs between the weights and activations. Increasing the numerical similarity between the two training data can alleviate the floating-point loss of numerical computation. Therefore, we normalize the weights according to the distribution of activations of the previous layer. Specifically, we normalize the weights with the same learnable parameter R_{max}. We call this normalization technique activation-aware normalization. The formula is as follows:

$$\hat{W} \leftarrow R_{max} * \frac{W - \mu_W}{\sqrt{\sigma_W^2 + \epsilon}}, \tag{1}$$

where μ_W and σ_W^2 are the mean and variance of weights W, ϵ is the minimal amount introduced to prevent division by zero, $\hat{W}$ is the normalized weights. Afterwards, the normalized weight will replace the original weights and participate in all forward and backward propagation.

4.3 Range-Aware Loss Scaling for Gradients

In backward propagation, the gradients of the previous layer and weights are computed by the gradients, activations, and weights of the current layer. Therefore, constructing a normalized gradients that has the same distribution as activations and weights is also another part to improve the numerical accuracy of computation. Therefore, this paper proposes a range-aware loss scaling and introduce a loss scaling factor S to adjust the distribution of the gradients to match the distribution of activations and weights.

Figure 2 illustrates the process of range-aware loss scaling. First, the loss obtained from forward propagation can be scaled by multiplying by the loss scaling factor S. Then, the backward propagation deduces based on the scaled gradients and the scaled weight gradients. Weights are then updated by applying re-scaling to the scaled weight gradients. In addition, the variance of the scaled gradients is counted and compared with the learnable variance R_{max} to adjust the loss scaling factor S.

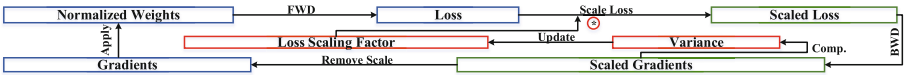

Fig. 2. Evaluation accuracy of four training approaches.

Specifically, the workflow of adjusting loss scaling factor S consists of three steps: ❶ loss scaling factor S starts from a relatively high value (e.g., $S \leftarrow 224$) because the gradient is generally small, and then the variance of the gradients is checked over iterations; ❷ If the variance of the gradients is close to R_{max} within a threshold (e.g., 10% difference), the scaling factor will not be adjusted and training continues; if the variance of the gradients is much larger than R_{max}, the loss scaling factor S will be halved to reduce the data distribution; Otherwise, the loss scaling factor S will be doubled to build a wider data distribution; ❸ the adjustment process will go throughout the whole training process because its overhead is almost negligible due to only a few multiplications are added.

5 Adaptive Layer Modifier in TrainBF

In this section, we discuss the opportunity of applying these normalization techniques to few layers with acceptable overhead and sufficient accuracy.

5.1 Sensitivity Study

We use two data formats, ie, TF32 and BFloat16, and apply the normalization techniques to different layers to study its affects on model accuracy and overhead. We run eight models in Mlperf benchmark [19] on one NVIDIA A100 GPU using two data formats and apply normalization techniques to each layer

separately. We use the exact same initialization values for both two data formats and treat the output of each layer of using TF32 as the ground-truth to calculate the computational error of using BFloat16. The computational error ε_L of each activations A_L in layer L between TF32 and BFloat16 can be expressed as $\varepsilon_L = \sum(A^i_L\_TF32 - A^i_L\_BF16)^2 / \sum(A^i_L\_TF32)^2$, where $A^i_L\_TF32$ and $A^i_L\_BF16$ represent the activations in TF32 and BFloat16, respectively.

Results reveals that applying normalization to each layer always comes with overhead but not always bring the same benefit to computational error. For example, applying normalization to layer 7 in ResNet-50 model has a computational error of 0.926%, which is much better than not using normalization that has a computational error of 2.754%. While adding normalization to two more layers (e.g., layer 1 and 15) leads to a similar computational error of 0.927%. Nevertheless, adding more normalization operations incurs larger overhead. In this same example, the throughput of using normalization on three layers is 74.86% of that of using normalization on one layer. Hence, blindly applying normalization to all the layers in DNN models may result in unacceptable overhead.

We further analyze the collected results of throughput and computational error in all eight models and summarize some interesting observations.

- **Observation 1:** Using normalization to too many layers largely reduces the throughput of model training.
- **Observation 2:** Using normalization for each layer does not have the same effect on reducing computational error. It strongly depends on where does the normalization occur in the model.
- **Observation 3:** Inappropriate use of too many normalization operations may not be necessary. Applying a small number of normalization operations can also achieve the optimal throughput while meeting the accuracy requirement of numerical computation.
- **Observation 4:** Computational error gradually propagates backwards. There is no point in correcting error at the very beginning or at the end of the model.

5.2 Adaptive Layer Modifier

Driven by these observations, we introduce a lightweight and adaptive layer modifier to apply normalization and maximize training throughput. Algorithm 2 depicts its workflow. Layer modifier first avoids applying normalization to the first f and last l layers because of Observations 1 and 4 (Line 6), where f and l are predefined values and are typically 5% of the number of layers. Then, layer modifier collects activations of each layer using TF32/Float32 and BFloat16 formats to calculate the computational error between them (Line 8–12). Layer modifier chooses the layer with the largest computational error and applies normalization to it (Line 13–15). Next, the computational error of the last layer between in two formats is tested (Line 16), and new normalization operations continue to be added until the computational error is less than a threshold (Line 17–18). The algorithm happens only once before training, therefore, its overhead has a negligible impact on end-to-end training time.

Algorithm 2. Lightweight and Adaptive Layer Modifier

Input: DNN model with N layers (L_1, L_2, ... ,L_N)
Input: A batch of testing dataset B, first layers f, last layers l
Output: A set of layers S that need to be normalized
All layers in DNN model $M \leftarrow \{1, 2, ..., N\}$,
An empty set of errors $E \leftarrow \{\}$, An empty set of layers $S \leftarrow \{\}$
Remove the first f and last l layers from set M
while true **do**
  for $i \in$ set M **do**
    Obtain activations A_i^{FP32} of layer L_i using data B with TF32 format
    Obtain activations A_i^{BF16} of layer L_i using data B with BFloat16 format
    Compute the computational error E_i between A_i^{FP32} and A_i^{BF16}
    Keep the computational error of each layer $E \leftarrow E + E_i$
  Choose the one with the greatest error in set E with the index of o
  Remove o from set M and add o to set S
  Apply normalization techniques to layer o
  Compute the final error $Final\_E$ between TF32 and BFloat16 format using data B
  if $Final\_E < threshold$ **then**
    Return a set of layers S

6 Efficient Parallel Strategy in TrainBF

TrainBF is also a work aiming at efficiently training DNN models on AI accelerators that have high parallelism and large memory. We propose an *efficient parallel strategy* to train DNN models using multiple execution streams. In addition, this strategy maintains the same batch size as single precision training to avoid the non-convergence and gradient explosion problem.

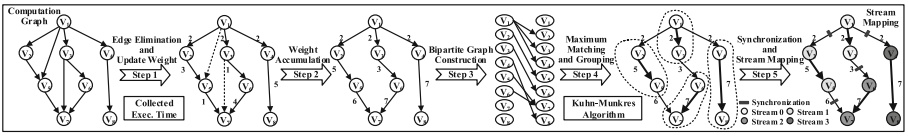

Fig. 3. Evaluation accuracy of four training approaches.

We propose an efficient parallel strategy to maximize memory usage, it is divided into two parts: the first one is an operator-to-stream mapping algorithm, where the input is the compiled computational graph of the model (such as TorchScript graph in PyTorch), and the output is the mapping between operators and execution streams; The second one is a runtime algorithm that collects the execution time of each operator and controls memory allocation of each stream.

Figure 3 describes the execution flow of the operator-to-stream mapping algorithm. At step ❶, we first eliminate the unnecessary edges with the minimum equivalent graph to avoid repeated and progressive data dependencies. For example, there are data dependencies from V_1 to V_2 and V_2 to V_5, so the data dependencies from V_1 to V_5 are repeated and can be removed. In addition, we collect the execution time of each operator in the previous iteration and use them as the weight of edges. Specifically, the weight of each edge is equal to the execution time of the outgoing node, because the incoming node must wait for all the incoming edges to complete before starting. At step ❷, the weight of each edge is accumulated with the weights of all the edges in the max-flow augmentation path

to obtain the weight accumulation graph, which represents the minimum execution time for fully parallel execution. At steps ❸ and ❹, a weighted bipartite graph is constructed based on the weight accumulation graph, and its maximum matching is calculated by a typical graph algorithm, namely Kuhn-Munkres algorithm [33]. Then the grouping strategy minimizes the sum of weighted data dependence between groups, thereby minimizing the sum of the waiting time of each group. At step ❺, synchronization points are added to each edge between each group to ensure the correctness of the execution order, and each group is assigned to a different execution stream.

After getting the operator-to-stream mapping, we start all execution streams simultaneously at the beginning of training to maximize parallelism. However, each operation performed in a different execution stream consumes a certain amount of independent memory resources, and executing multiple memory-consuming operators in different streams simultaneously could lead to Out-of-Memory(OOM) issue. Therefore, we enable a memory table to check whether the memory overflows before each operator is launched. In addition, the execution time of each operator is recorded and passed to the operator-to-stream mapping algorithm to update the weight of the computational graph.

7 Evaluation

7.1 Experimental Setup

Platforms and Formats: We evaluate TrainBF on three architectures, as shown in Table 1. Two of them are GPU-based platforms equipped with NVIDIA A100 GPU (A100 in short) and AMD MI100 GPU (MI100 in short), respectively. The third is an AI accelerator-based platform, SambaNova SN10-8 (SambaNova in short). A100 and MI100 GPUs support Float32, Float16, and BFloat16 formats. A100 GPU also supports TF32 [3]. SambaNova supports Float32 and BFloat16 formats.

Table 1. Evaluated hardware

	NVIDIA GPU	AMD GPU	AI accelerator
Core	Tesla A100 40 GB 56 SMs @1328 MHz	AMD Instinct MI100 120 Compute Units @1502 MHz	SambaNova SN10-8 640 PCUs 640 PMUs
Caches	L2: 40 MB	L2: 8 MB	On-chip: 300MB
Memory	40 GB HBM2	32 GB HBM2	12TB DDR4
Bandwidth	1555 GB/s	1200 GB/s	150TB/s

Table 2. DNN models, datasets, and configurations

DNN Model	Field	Dataset	Epoch	Throughput Unit
Resnet50	Image Recognition	ILSVRC2012	90	Images per second
VGG19	Image Recognition	ILSVRC2012	100	Images per second
U-Net	Image Segmentation	Brain MRI Kaggle3m	30	Images per second
Social-LSTM	Trajectory Prediction	Trajnet++	100	Sequences per second
GCN	Graph Computation	Cora Dataset	200	Items per second
UNO	HPC model	CCLE Dataset	50	Items per second

Dataset and Models: We use six DNN models with a public dataset that cover a wide range of CNN, RNN, GNN, and scientific models. The details of the models are summarized in Table 2. *Epoch* represents the number of epochs trained before obtaining the final model accuracy, *Throughput Unit* is the unit of throughput of each model during training. We use different batch sizes on different platforms to fill all available memory to maximize memory utilization.

Implementation and Baselines: This work is implemented based on PyTorch 1.11.0. We implement the three customized normalization techniques as three new modules in PyTorch. The statistics of modification of TrainBF given by git diff are 24 files changed, 1535 insertions (+), and 349 deletions (−).

We compare TrainBF with three solutions:

❶ A single-precision solution: pure Float32 or TF-32 training.

❷ A mixed-precision solution: Automatic Mixed Precision (AMP) with Float16 [23].

❸ A half-precision solution: pure BFloat16 training.

For a fair comparison, we compare TrainBF with AMP using Float16 on A100 since it provides the same theoretical performance for both Float16 and BFloat16. For MI100 and SambaNova, neither platform supports the same performance for Float16 and BFloat16, thus we compare the throughput and accuracy of TrainBF with the throughput of Float32 training and the accuracy of BFloat16 training, respectively. In addition, all six models are tested on A100 and MI100. For SambaNova, only two models (U-Net and UNO) are tested, because the support for LSTM and some kernels will not be released until Q4 2023.

7.2 Throughput and Accuracy

Figure 4 shows throughput and accuracy on all platforms. We run all models on A100 and MI100 and two models on SambaNova due to its limited support.

Figure 4 shows that TrainBF performs much better than the state-of-the-art training approaches. Specifically, for A100, TrainBF introduces 1.74×, 1.52×, 1.61×, 1.31×, 1.46×, 1.08× throughput improvement on six DNN models respectively, compared to TF32 training, with only 0.48% accuracy degradation on average, which is far below the accuracy loss of 1.5% that users can tolerate for training [25]. TrainBF also introduces 1.31×, 1.15×, 1.09×, 1.13×, 1.37×, 1.67× throughput improvement, compared to AMP with Float16, with almost the same accuracy. TrainBF improves the final accuracy by 15.7% on average and up to 45.8% on UNO model, compared to BFloat16 training.

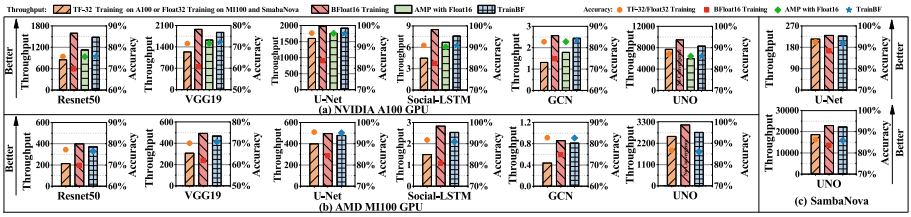

Fig. 4. Throughput and accuracy using four training methods on six models with three different hardware platforms.

For MI100, TrainBF introduces 1.63×, 1.40×, 1.83×, 1.42×, 1.53×, 1.04× throughput improvement on six DNN models respectively, compared to Float32

training, with only 0.52% accuracy loss on average. TrainBF improves the finial accuracy by 13.9% on average, compared to BFloat16 training.

For SambaNova platform, TrainBF introduces 1.15× and 1.18× throughput improvement on U-Net, UNO models, compared to Float32 training, with the accuracy loss of 0.32% on average. TrainBF improves the final accuracy by 3.59% on average, compared to BFloat16 training.

We have the following three observations: (1) TrainBF brings larger benefits to CNN models, because matrix multiplication as the main computation in CNN models can take full advantage of the high performance of BFloat16 format. (2) For RNN and GNN models, MI100 has higher speedup than A100, because these models are memory intensive. The amount of data accessed is greatly reduced by using BFloat16, which eliminates the bottleneck of lower memory bandwidth on MI100 compared to A100. (3) For SambaNova, TrainBF achieves almost the same throughput as BFloat16 training while maintaining the Float32 accuracy.

7.3 Breakdown for Accuracy Improvement

To quantify **the contribution of three normalization techniques to accuracy improvement**, i.e., (a) central and range-maximized normalization, (b) activation-aware normalization, and (c) range-aware loss scaling, we apply the three techniques one after another. The results in Fig. 5 are normalized by using the accuracy of applying all of the three techniques.

We have three observations. (1) The central and range-maximized normalization is very effective and accounts for 48.3% on average in improving model accuracy across all models, because this normalization is the cornerstone of reducing computational error, thus enabling more opportunities for all subsequent techniques. (2) The activation-aware normalization is very effective (52.7% on average) for the RNN model (e.g., social-LSTM) because a large number of small matrix multiplication are computed in RNN training, and the normalized weight could prevent the error of small matrix from propagating to the following computations, thereby avoiding greater accuracy loss. (3) The range-aware loss scaling contributes 33.1% on average to GNN and scientific models (e.g., GCN and UNO), because the loss in these models varies greatly, making the distribution range of gradients very unstable without scaling.

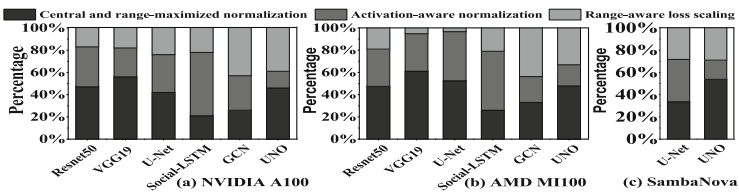

Fig. 5. Quantifying the contributions of three normalization to accuracy improvement.

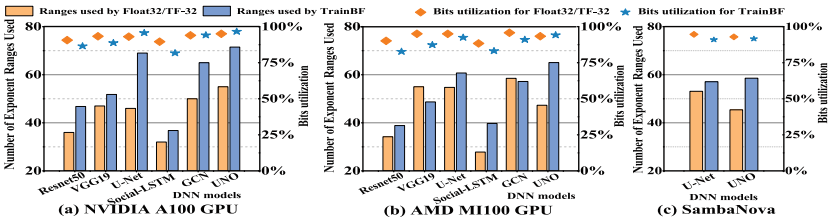

Fig. 6. Number of exponent ranges used and bits utilization on three platforms.

7.4 Effectiveness of Three Modules in TrainBF

Quantifying Bits Utilization. We use *number of exponent ranges used* to quantify the bits utilization. Results are shown in Fig. 6. With TrainBF, the average number of exponent ranges used on all models is improved from 41.4 to 57.6 on A100, 40.7 to 51.3 on MI100, and 49.7 to 57.6 on SambaNova. With TrainBF, the average bits utilization on all models is 92.7% on A100, 88.4% on MI100, and 91.5% on SambaNova. The bits utilization of BFloat16 in TrainBF is very close to the bits utilization of TF32/Float32 in single precision training and even exceeds by 1.7% and 3.5% on average on A100 and MI100 for GCN and UNO models. Based on the improvement of bits utilization, there is a large increase in computational accuracy, further improving model accuracy.

Quantifying Learnable Parameter R_{max}. TrainBF uses the learnable parameter R_{max} to controls the variance of normalized output. In our experiments, R_{max} is initialized to 1 and reaches 2.5 in the first 25% of the training process for most models, which implies that the primary (95%) data range of activations and weights are 1.45 times larger than the initial data range. Among the eight DNN models we evaluate, R_{max} is stable for all three CNN models and three scientific models in the last 75% of the training process, while R_{max} changes more drastically in the other two models, namely social-LSTM and GCN. The main reason is that the data distribution of gradients on social-LSTM and GCN differ greatly over epochs in model training, so R_{max} is constantly tuned to find the optimal value that matches the distribution of activations.

Quantifying Efficient Parallel Strategy. TrainBF leverages the memory space saved by half-precision format to parallelize the execution of training operators and increase memory usage. Compared with TrainBF without an efficient parallel strategy, TrainBF brings 1.13× and 1.29× performance improvement on A100 and MI100, because the execution strategy of closed-source SambaNova cannot be modified. Compared with the naive implementation of BFloat16 training, our efficient parallel strategy recognizes independent operators and executes them simultaneously, and results show that the memory usage is 65.32% and 74.17% higher than naive implementation on A100 and MI100, respectively.

7.5 Overhead Analysis

We explore **the overhead of TrainBF** by comparing the throughput of TrainBF without the efficient parallel strategy and that of BFloat16 training in Fig. 4. The throughput of BFloat16 training represents the optimal training performance regardless of model accuracy in single-stream execution. After applying these normalization techniques to selected layers, the calculation of throughput will include all of the runtime overhead. Compared with the throughput of BFloat16 training, TrainBF introduces an average throughput degradation of 3.38%, 7.91%, and 9.67% on A100, MI100, and SambaNova, respectively. Obviously, A100 and MI100 have lower overhead, because the normalization operations can be merged by fusion optimization in GPU implementation.

8 Related Work

Reduced Precision Training: Using reduced precision for DNN training has been an active topic of research [6,9–12,28,34]. Seide et al. [24] were able to reduce the precision of gradients to one bit using Stochastic Gradient Descent. However, these works mainly focus on a small number of models and lack generality to apply to a wider range of DNN models.

Mixed Precision Training: Mixed precision training demonstrates a broad variety of DNN applications involving deep networks and larger datasets with minimal loss compared to baseline FP32 results. Micikevicius et al. [21] showed that Float16/Float32 mixed precision with autocasting and loss scaling can achieve near-SOTA accuracy. The only concern is about performance improvement by using Float16. TrainBF leverages BFloat16 format to avoid such overhead and maintain SOTA accuracy.

Normalization: Normalization techniques are essential for improving the generalization of DNN models [29–31]. Dmitry et al. [26] constructed instance normalization to prevent instance-specific mean and covariance shifts. Yuxin et al. [27] proposed group normalization to normalize features within each group. None of these are designed to eliminate computational error, which is the main goal of this paper.

9 Conclusion

BFloat16, as a typical half-precision format, has been neglected in recent AI accelerators. This paper designs a new training approach, which includes three normalization techniques, an adaptive layer modifier, and an efficient parallel strategy to avoid accuracy loss and improve hardware utilization. Results show that our approach yields better throughput than the state-of-the-art training approaches. We expect more data formats can be inspired by our approach.

Acknowledgment. This research was funded in part by and used resources at the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC02-06CH11357.

References

1. Blinn, J.F.: Floating-point tricks. IEEE Comput. Graphics Appl. **17**(4), 80–84 (1997)
2. Burgess, N., Milanovic, J., Stephens, N., Monachopoulos, K., Mansell, D.: BFloat16 processing for neural networks. In: 2019 IEEE 26th Symposium on Computer Arithmetic (ARITH), pp. 88–91. IEEE (2019)
3. Choquette, J., Gandhi, W., Giroux, O., Stam, N., Krashinsky, R.: NVIDIA A100 tensor core GPU: performance and innovation. IEEE Micro **41**(2), 29–35 (2021)
4. contributors, W.: BFloat16 floating-point format (2021). https://en.wikipedia.org/wiki/Bfloat16_floating-point_format
5. Das, D., et al.: Mixed precision training of convolutional neural networks using integer operations. arXiv preprint arXiv:1802.00930 (2018)
6. Emani, M., et al.: A comprehensive evaluation of novel AI accelerators for deep learning workloads. In: 2022 IEEE/ACM International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS), pp. 13–25. IEEE (2022)
7. Franchi, G., Bursuc, A., Aldea, E., Dubuisson, S., Bloch, I.: TRADI: tracking deep neural network weight distributions. In: Vedaldi, A., Bischof, H., Brox, T., Frahm, J.-M. (eds.) ECCV 2020. LNCS, vol. 12362, pp. 105–121. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-58520-4_7
8. Gupta, S., Agrawal, A., Gopalakrishnan, K., Narayanan, P.: Deep learning with limited numerical precision. In: International Conference on Machine Learning, pp. 1737–1746. PMLR (2015)
9. He, X., Chen, Z., Sun, J., Chen, H., Li, D., Quan, Z.: Exploring synchronization in cache coherent manycore systems: a case study with xeon phi. In: 2017 IEEE 23rd International Conference on Parallel and Distributed Systems (ICPADS), pp. 232–239. IEEE (2017)
10. He, X., et al.: Enabling energy-efficient DNN training on hybrid GPU-FPGA accelerators. In: Proceedings of the ACM International Conference on Supercomputing, pp. 227–241 (2021)
11. He, X., Sun, J., Chen, H., Li, D.: Campo: {Cost-Aware} performance optimization for {Mixed-Precision} neural network training. In: 2022 USENIX Annual Technical Conference (USENIX ATC 22), pp. 505–518 (2022)
12. He, X., Yao, Y., Chen, Z., Sun, J., Chen, H.: Efficient parallel A* search on multi-GPU system. Futur. Gener. Comput. Syst. **123**, 35–47 (2021)
13. Ioffe, S., Szegedy, C.: Batch normalization: accelerating deep network training by reducing internal covariate shift. In: International Conference on Machine Learning, pp. 448–456. PMLR (2015)
14. Jia, X., et al.: Highly scalable deep learning training system with mixed-precision: training ImageNet in four minutes. arXiv preprint arXiv:1807.11205 (2018)
15. Johnson, J.: Rethinking floating point for deep learning. arXiv preprint arXiv:1811.01721 (2018)
16. Johnston, J.T., et al.: Fine-grained exploitation of mixed precision for faster CNN training. In: 2019 IEEE/ACM Workshop on Machine Learning in High Performance Computing Environments (MLHPC), pp. 9–18. IEEE (2019)
17. Kuchaiev, O., Ginsburg, B., Gitman, I., Lavrukhin, V., Case, C., Micikevicius, P.: OpenSeq2Seq: extensible toolkit for distributed and mixed precision training of sequence-to-sequence models. In: Proceedings of Workshop for NLP Open Source Software (NLP-OSS), pp. 41–46 (2018)

18. Kuchaiev, O., et al.: Mixed-precision training for NLP and speech recognition with openseq2seq. arXiv preprint arXiv:1805.10387 (2018)
19. Mattson, P., et al.: MLPerf training benchmark. Proc. Mach. Learn. Syst. **2**, 336–349 (2020)
20. Mellempudi, N., Srinivasan, S., Das, D., Kaul, B.: Mixed precision training with 8-bit floating point. arXiv preprint arXiv:1905.12334 (2019)
21. Micikevicius, P., et al.: Mixed precision training. arXiv preprint arXiv:1710.03740 (2017)
22. Mishra, A., Nurvitadhi, E., Cook, J.J., Marr, D.: WRPN: wide reduced-precision networks. arXiv preprint arXiv:1709.01134 (2017)
23. PyTorch: Automatic Mixed Precision package (2022). https://pytorch.org/docs/stable/amp.html. Accessed 1 Aug 2022
24. Seide, F., Fu, H., Droppo, J., Li, G., Yu, D.: 1-bit stochastic gradient descent and its application to data-parallel distributed training of speech DNNs. In: Fifteenth Annual Conference of the International Speech Communication Association. Citeseer (2014)
25. Sze, V., Chen, Y.H., Yang, T.J., Emer, J.S.: Efficient processing of deep neural networks: a tutorial and survey. Proc. IEEE **105**(12), 2295–2329 (2017)
26. Ulyanov, D., Vedaldi, A., Lempitsky, V.: Instance normalization: the missing ingredient for fast stylization. arXiv preprint arXiv:1607.08022 (2016)
27. Wu, Y., He, K.: Group normalization. In: Proceedings of the European conference on computer vision (ECCV), pp. 3–19 (2018)
28. Xie, Z., Dong, W., Liu, J., Liu, H., Li, D.: Tahoe: tree structure-aware high performance inference engine for decision tree ensemble on GPU. In: Proceedings of the Sixteenth European Conference on Computer Systems, pp. 426–440 (2021)
29. Xie, Z., Dong, W., Liu, J., Peng, I., Ma, Y., Li, D.: MD-HM: memoization-based molecular dynamics simulations on big memory system. In: Proceedings of the ACM International Conference on Supercomputing, pp. 215–226 (2021)
30. Xie, Z., Liu, J., Li, J., Li, D.: Merchandiser: data placement on heterogeneous memory for task-parallel HPC applications with load-balance awareness (2023)
31. Xie, Z., Tan, G., Liu, W., Sun, N.: IA-SpGEMM: an input-aware auto-tuning framework for parallel sparse matrix-matrix multiplication. In: Proceedings of the ACM International Conference on Supercomputing, pp. 94–105 (2019)
32. Zamirai, P., Zhang, J., Aberger, C.R., De Sa, C.: Revisiting BFloat16 training. arXiv preprint arXiv:2010.06192 (2020)
33. Zhu, H., Zhou, M., Alkins, R.: Group role assignment via a Kuhn-Munkres algorithm-based solution. IEEE Trans. Syst. Man Cybern.-Part A: Syst. Hum. **42**(3), 739–750 (2011)
34. Zvyagin, M., et al.: GenSLMs: genome-scale language models reveal SARS-CoV-2 evolutionary dynamics. bioRxiv, p. 2022–10 (2022)