Campo: Cost-Aware Performance Optimization for Mixed-Precision Neural Network Training

Xin He, *CSEE, Hunan University & Xidian University;* Jianhua Sun and Hao Chen, *CSEE, Hunan University;* Dong Li, *University of California, Merced*

https://www.usenix.org/conference/atc22/presentation/he

This paper is included in the Proceedings of the 2022 USENIX Annual Technical Conference.

July 11–13, 2022 • Carlsbad, CA, USA

978-1-939133-29-8

Open access to the Proceedings of the 2022 USENIX Annual Technical Conference is sponsored by

Campo: Cost-Aware Performance Optimization for Mixed-Precision Neural Network Training

Xin He
CSEE, Hunan University & Xidian University

Jianhua Sun
CSEE, Hunan University

Hao Chen
CSEE, Hunan University

Dong Li
University of California, Merced

Abstract

Mixed precision training uses a mixture of full and lower precisions for neural network (NN) training. Applying mixed precision must cast tensors in NN from float32 (FP32) to float16 (FP16) or vice versa. The existing strategy greedily applies FP16 to performance-critical operations without quantifying and considering the casting cost. However, we reveal that the casting cost can take more than 21% of NN operation execution time, and in some cases surpasses the performance benefit of using low precision. In this paper, we introduce Campo, a tool that improves performance of mixed-precision NN training with the awareness of casting costs. Campo is built upon performance modeling that predicts the casting cost and operation performance with low precision, and introduces a cost-aware graph rewriting strategy. Campo is user-transparent, and enables high performance NN training using mixed precision without training accuracy loss. Evaluating Campo with six NN models, we show that compared to TensorFlow using TF_AMP (a state-of-the-art performance optimizer for mixed precision training from Nvidia), Campo improves training throughput by 20.8% on average (up to 24.5%) on RTX 2080 Ti GPU and by 20.9% on average (up to 23.4%) on V100 GPU, without training accuracy loss. Because of using the cost-aware mixed precision training, Campo also improves energy efficiency by 21.4% on average (up to 24.2%), compared to TensorFlow using TF_AMP.

1 Introduction

Training Neural network (NN) can be resource-demanding: it consumes many compute cycles and requires high memory bandwidth and capacity. One promising approach to lower the resource requirements is to use mixed precision training [1]. Mixed precision is a computational method using a mixture of full and lower precisions. Mixed precision can deliver significant computational speedup by executing operations in a lower precision format as much as possible, while storing critical information in the full-precision format to preserve task-specific accuracy. The mixed precision training has shown significant speedup over the single (full) precision training on a variety of NN [2–4], especially with tensor cores (TC) available on some GPU architectures.

In NN training frameworks like TensorFlow and PyTorch, the mixed precision training is implemented via a graph rewrite process [5]. This process casts tensors referenced in certain operations in NN from float32 (FP32) to float16 (FP16), or vice versa. This process adopts a greedy strategy that applies FP16 execution to performance-critical operations, most of which are matrix multiplication and convolution, based on an implicit assumption that using low precision *always* leads to performance improvement, and hence the casting cost can be always justified.

However, our detailed performance analysis reveals that the above common assumption is not true. We observe that the casting cost can take more than 21% of operation execution time, depending on the input tensor size of the operation using mixed precision. In some cases (e.g., the operation `MatMul` with an input data size of (64, 1001, 1001, 2048)), the casting cost surpasses the performance benefit of using low precision. As a result, using mixed precision training (even with TC) may not be performance-beneficial and even leads to performance loss (22.7% in the example of `MatMul`). Hence, the casting cost must be considered and quantified when deciding precision for operations.

Deciding whether using low precision for an operation is performance-beneficial is challenging, because the operation time and casting cost are affected by input data size, whether TC is used, and performance characterization of operation (e.g., memory access pattern and compute intensity). Also, using low precision should not impact NN model accuracy. Hence, making the decision of using low precision is a multi-dimensional problem.

In this paper, we introduce a cost-aware performance optimization tool, named Campo, aiming to improve performance of mixed-precision NN training with the awareness of casting costs. Campo assigns the low or full computation precision to each operation in NN to maximize performance while preserving the NN model accuracy. Campo is built upon perfor-

mance modeling that predicts the casting cost and operation performance with low precision. The performance modeling is operation-specific and captures events (such as L2 cache misses and global loads/stores on GPU) critical to the performance of low precision and collected through dynamic profiling in full precision. Using dynamic profiling, the performance modeling is also able to capture the impact of input data size on performance.

Leveraging the performance modeling, Campo introduces a cost-aware graph rewriting strategy. This strategy avoids applying low precision to those operations that cannot get performance benefit, and minimizes the casting cost when applying low precision to a group of operations. Furthermore, Campo does not impact NN model accuracy, because it only applies low precision to those operations identified as numerical safe by the traditional algorithm for low precision assignment. In addition, some operations can benefit from low precision but cannot run on TC because their input data sizes cannot meet the requirement of TC. For those operations, Campo pads the input tensors without programmer participation to maximize the utilization of TC for high performance.

We summarize major contributions as follows.

- We conduct a comprehensive performance characterization on operations in NN training and quantify casting costs. In contrast to the traditional methods that decide precision assignment without considering the casting cost, we reveal that the casting cost can outweigh the performance benefit of using low precision. This observation is unprecedented.
- We develop novel and practical performance modeling to predict casting cost and the performance of operations in low precision.
- We propose Campo, a performance optimization tool that enables high-performance mixed precision training without losing training accuracy. Campo uses a graph traverse algorithm and performance modeling to assign low or high precision to each operation.
- We implement Campo within TensorFlow, and evaluate it with six NN models on Nvidia GeForce RTX 2080 Ti and V100 GPUs. Our evaluation shows that compared to TensorFlow using TF_AMP (a state-of-the-art performance optimizer for mixed precision training from Nvidia), Campo improves training throughput by 20.8% on average (up to 24.5%) on RTX 2080 Ti and by 20.9% on average (up to 23.4%) on V100, without losing training accuracy. Because of using cost-aware mixed precision training, Campo improves energy efficiency by 21.4% on average (up to 24.2%), compared with TensorFlow using TF_AMP.

2 Background

2.1 Mixed Precision Training

A dominant programming paradigm, commonly adopted by machine learning frameworks such as TensorFlow and PyTorch, is to represent an NN model as a static dataflow graph, where computation functions (e.g., `Conv2D`, and `MatMul`) in the NN model are associated with nodes in the graph, and input and output tensors of the computation map to edges. The architecture of the NN model (i.e., the dataflow graph) is defined using symbolic expressions by the programmer; The common computation functions are defined as *operations* by the machine learning framework.

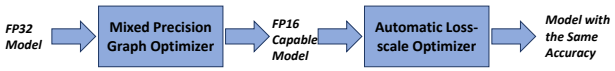

Figure 1: The workflow of mixed precision training.

The mixed precision training makes precision assignment decisions per node. Some nodes (i.e., operations) in the dataflow graph use full precision (i.e., FP32), and those nodes are essential to maintain training accuracy. Other nodes use lower precision, which is useful to save memory capacity and bandwidth and enable faster math operations (especially on GPUs with TC support). As a result, the mixed precision training can harvest the best of both worlds: maintaining training accuracy and having fast execution.

In TensorFlow and PyTorch, only FP16 is considered as lower precision for the mixed precision training because of FP16's commonality in various GPU architectures, although some GPU architectures (such as Turing and Amphere) support other lower precisions, such as INT8 and INT4. Like TensorFlow and PyTorch, we only consider FP16 as lower precision in this paper, because of FP16's commonality.

Figure 1 depicts the workflow of using the mixed precision training in TensorFlow and PyTorch. In general, it includes two steps: (1) identifying which nodes should be changed to FP16 and inserting casts between FP32 nodes and FP16 nodes by a mixed precision graph optimizer, and (2) adding loss scaling to preserve small gradient values by an automatic loss-scale optimizer. We focus on (1) in this paper.

The mixed precision graph optimizer decides the assignment of low precision to operations by classifying operations into multiple lists based on operation's numerical safety. The numerical safety refers to how an NN model's quality is affected by the use of low precision. An operation is numerical unsafe, if using low precision during the operation execution leads to worse training accuracy, compared with using FP32. In TensorFlow, there are four lists, discussed as followed.

- ***Allowlist***: operations (e.g., `MatMul` and `Conv2D`) in this list are considered numerically-safe for execution in FP16, and also performance-critical. These operations are always converted to use FP16.
- ***Denylist***: operations (e.g., `Exp` and `SoftMax`) in this list are considered numerically-dangerous in FP16 and their effects may also be observed in a downstream operation node. For example, using FP16 in the operation sequence of `Exp` ->`Add`, the `Add` is unsafe due to the unsafe `Exp`.

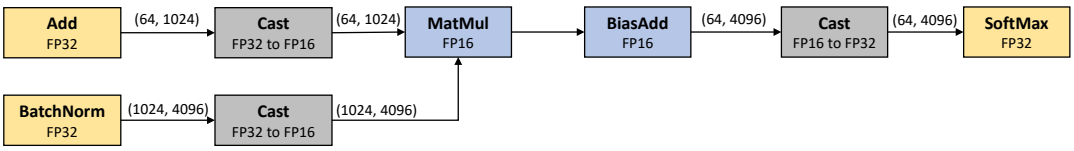

Figure 2: A snippet of the dataflow graph from BERT using mixed precision. The tuple on each edge represents a tensor with its shape (i.e., the size of each dimension).

- ***Inferlist***: operations (e.g., `BiasAdd`) in this list are considered numerically-safe in FP16, which may be, however, made unsafe by an upstream *denylist* operation.
- ***Clearlist***: operations (e.g., `Max` and `Min`) in this list do not have numerically-significant effects in the sense that they can be executed either in FP16 or in FP32.

Compared with TensorFlow, PyTorch use three lists because the operations in the *clearlist* and the *inferlist* are classified into a single list.

Using the above lists, the mixed precision graph optimizer uses low precision for an operation, if any of the following three conditions is true: (1) The operation is in the *allowlist*; (2) the operation is in the *clearlist*, and its immediate ancestor(s) and immediate descendent(s) are using low precision; (3) the operation is in the *inferlist* and there is no upstream *denylist* operation. The mixed precision graph optimization works online by re-writing the dataflow graph via inserting casts before the iterative training of NN model happens.

Figure 2 shows a mixed-precision graph snippet taken from BERT [6] (a transformer-based model). To enable FP16 for `MatMul` and `BiasAdd` while ensuring numerical safety, two FP32-to-FP16 cast operation nodes and one FP16-to-FP32 cast operation node are inserted into the graph. Take the FP32-to-FP16 cast operation node with the input data size (64, 1024) as an example. The number of scalar elements in the input data (tensor) is 65536 (i.e., 64×1024) in this example.

2.2 Tensor Core Acceleration

Since Volta architecture, Nvidia introduces specialized hardware arithmetic units into its GPU products, called Tensor Cores (TC). Compared to regular CUDA cores, TC is more performant and energy-efficient. TC is used to accelerate FP16 matrix multiplication and convolution operations. In TensorFlow, these operations refer to `MatMul`, `Conv2DBackpropFilter`, `Conv2DBackpropInput` and `Conv2D`, which are usually the most fundamental and time-consuming operations in NN models.

TC is automatically activated to run an operation when two conditions are met: (1) the operation is either matrix multiplication or convolution using FP16, and (2) the input tensors of the operation satisfy the shape requirements. For (2), TC requires certain dimensions of the tensor to be a multiple of 8. If the condition (1) is met, we say the operation is a *TC candidate*. Such an operation can run on regular CUDA cores with low precision when the condition (2) is not met.

Besides the above discussion on the mixed precision training and TC, we target on those NN models whose dataflow graphs are static, which indicates that the dataflow graph does not change its structure across training samples and hence each training step goes through the exactly same computation graph. This implies that once the training batch size is determined, the input data size and shape (i.e., the size of each dimension) of operations are known before the training happens. Such NN models are very common and have been a research target in many recent efforts [7–13].

3 Observation and Motivation

To motivate the design of Campo, we characterize the performance of operations under full precision and low precision, and study the casting cost. Table 1 shows the performance results for six operations. These operations can be commonly found in NN models. The first four operations in the table (`MatMul`, `Conv2D`, `Conv2DBackpropFilter`, and `Conv2DBackpropInput`) are candidates to run on TC. These four operations can easily account for most of the NN training time. For example, in ResNet50, the four operations take more than 90% of the total training time. The four operations fall into the *allowlist*, and hence are always converted for FP16 execution. The last two operations in the table (i.e., `BiasAdd` and `MaxPool`) fall into the *inferlist* and *clearlist*, respectively, of which the numerical precision chosen for the operations is usually context-dependent. We do not study the operations in the denylist because they are always executed in FP32. Besides the six operations in Table 1, we study other operations in the three lists, but do not show them in Table 1 for brevity.

We develop two microbenchmarks for each operation to run it with FP16 and FP32 respectively. The input data sizes for each operation are collected from Resnet-50, Inception3 and DCGAN by dlprof [14]. Among those input data sizes, some of them meet the TC requirement on tensor shape, and hence the corresponding operations run on TC. In our study, we use TensorFlow 1.15 and Nvidia RTX 2080 Ti GPU. We run each operation with each input data size 100 times and report the average result. In Table 1, "FP16 Exe. time" and "FP32 Exe. time" do not include casting cost.

Overall, we study the impact of data precision, TC, casting cost, and input data size on operation performance.

Table 1: Performance comparison of some of the representative operations in NN training.

NN Operations	Input Data Size	FP16 Exe. Time (ms)	FP16+Cast Exe. Time (ms)	FP32 Exe. Time (ms)	Using TC
MatMul	(2048, 8, 8, 1024)	0.312	0.353	0.323	yes
	(64, 1001, 1001, 2048)	0.412	0.524	0.427	no
	(2048, 1024, 1024, 1024)	0.414	0.584	0.888	yes
Conv2D	(64, 35, 35, 48)	2.707	2.795	3.664	yes
	(64, 147, 147, 32)	28.965	29.249	29.487	no
	(64, 299, 299, 3)	57.879	58.944	60.098	no
Conv2DBackpropFilter	(64, 299, 299, 3)	8.690	9.773	10.246	no
	(64, 149, 149, 32)	6.013	7.988	7.011	no
	(64, 35, 35, 192)	0.786	0.948	0.871	yes
Conv2DBackpropInput	(64, 37, 37, 96)	3.954	4.049	6.943	yes
	(64, 149, 149, 32)	15.561	16.828	15.696	no
	(64, 35, 35, 192)	5.234	5.939	10.060	yes
BiasAdd	(64, 1001, 1001)	0.252	0.317	0.255	no
	(64, 4096, 4096)	0.294	0.323	0.298	no
	(64, 9216, 9216)	0.299	0.342	0.311	no
MaxPool	(64, 35, 35, 288)	1.849	2.072	1.793	no
	(64, 17, 17, 768)	1.399	1.542	1.402	no
	(64, 8, 8, 2048)	0.825	1.128	0.981	no

1) ***Performance variance with different data precisions.*** Table 1 shows that across operations, FP16 consistently outperforms F32, regardless of using TC or not (and without consideration of casting cost). For example, despite not using TC, `MatMul` with FP16 performs slightly better (3.5%) than with FP32 for the input data size (64, 1001, 1001, 2048).

Furthermore, when the input shape meets the requirement of using TC, the performance gain of using FP16 over FP32 is more significant. For example, `Conv2DBackpropInput` with FP16 performs significantly better (48%) than with FP32 for the input data size (64, 35, 35, 192).

Observation 1. Without TC, using F16 leads to slightly better performance than using F32. Using TC for FP16 magnifies the performance benefit of F16.

2) ***Impact of input data size on performance gains from FP16.*** Training an NN model can invoke many instances of an operation in a training step. Different instances of the operation can use different input data sizes. Table 1 shows that as we change the input data size of an operation, the performance gain of using FP16 over using FP32 varies significantly. For example, for `MatMul` with FP16, the performance gain is 3.6% and 114.5% for the input data sizes (64, 1001, 1001, 2048) and (2048, 1024, 1024, 1024) respectively. In this example, such a large performance variance comes from whether TC is utilized. Even if TC is not utilized for TC candidate operations, we observe large performance variance across different input data sizes. For example, `Conv2DBackpropInput` with input data sizes (64, 149, 149, 32) and (64, 37, 37, 96) have 75.6% and 0.9% performance gains when using FP16.

The above observation holds true for the non-TC candidate operations as well. For example, `BiasAdd` with input data sizes (64, 9216, 9216) and (64, 1001, 1001) have 40.1% and 11.9% performance gain when using FP16.

The reason for the above result is because of smaller memory bandwidth consumption and smaller number of FP operations with smaller input data size, which offers less opportunity for FP16 to tap and improve performance.

Observation 2. The performance gain of using FP16 varies largely across input data sizes.

3) ***Impact of casting cost.*** Comparing "FP16+Cast Exe. Time" and "FP16 Exe. Time" in Table 1, we see that the cast operation introduces 3% - 29% overhead, diminishing the performance benefit of FP16. As a result, using FP16 can perform worse than FP32. For example, considering the casting cost, `MatMul` using a TC-satisfied input data size (2048, 8, 8, 1024) and a TC-unsatisfied input data size (64, 1001, 1001, 2048) with FP16 performs worse than with FP32 by 9.3% and 22.7% respectively, and the casting cost takes 11.6% and 21.4% of the operation execution time, which is large.

The casting cost stems from (1) the time to initialize the cast operation node in the dataflow graph, and (2) the time to do `bitcast` and numerical `truncation` for each scalar element in the input tensor as well as construct the output.

Observation 3. The cast operation introduces non-negligible overhead. Considering the casting cost, it is not always performance-profitable to convert FP32 to FP16 regardless of using TC or not.

In addition to the NVIDIA GeForce RTX 2080 Ti, we get the same three observations on Nvidia V100.

Implications of observations. The effectiveness of using low precision for an operation is impacted by input data size, casting cost, and the usage of TC. Optimizing the assignment of low precision to operations is a multi-dimensional problem, not just one dimensional problem as assumed in the existing solutions.

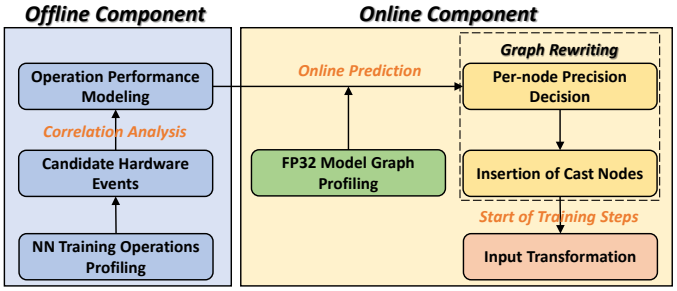

Figure 3: Overview of Campo.

4 Design

4.1 Overview

Campo includes an offline component and an online component, as illustrated in Figure 3. The offline component is used to build performance modeling to predict performance of operations in FP16. The performance modeling is used by the online component, and makes performance prediction using performance events (e.g., L2 cache misses and global memory load/store throughput) collected from operation execution in FP32. The performance modeling uses correlation analysis to decide which events are the most important for accurate performance prediction. The performance modeling is based on statistical regression modeling, which is built only once by the offline component but repeatedly used by the online component.

The online component includes graph profiling, graph traverse to make per-node precision decision, insertion of cast operation, and input transformation. The graph profiling uses one iteration to run operation nodes in FP32 and collect events needed by performance modeling. The graph traverse makes four passes on the dataflow graph of the NN model and uses performance modeling to decide if each operation should use FP16 or not based on the estimation of performance benefit and cost of using FP16. The input transformation pads the input tensors to make TC candidates meet the requirement of TC on input shape and thus improves the utilization of TC. The online component is lightweight and enables cost-aware mixed precision optimization with guarantee on performance improvement without training accuracy loss.

4.2 Performance Modeling

We build performance modeling to decide whether low or full precision should be employed for a given operation with a given input data size. The performance modeling predicts (1) the casting cost based on the input data size, and (2) execution time of the operation with low precision.

4.2.1 Predicting Casting Cost

The casting cost for a cast operation includes two parts: (1) the time to initialize the cast operation node in the dataflow graph, denoted by C_I, and (2) the time to do the conversion (i.e., the cost to do `bitcast` and numerical `truncation` for each scalar element in the input tensor as well as construct the output tensor), denoted by C_C.

C_I is modeled as a constant, because C_I comes from a couple of memory allocations and variable assignments for the object initialization of the cast operation node in the dataflow graph. C_C is proportional to the number of scalar elements in the input tensor of the cast operation node. This proportion is represented as a ratio, r. Given the same number of scalar elements, we observe that there is no performance difference between converting from FP16 to FP32 and from FP32 to FP16.

Hence, the casting cost is modeled in Equation 1, where *tensor_size* is the input data size of the cast operation node. Taking the operation `Add` with an input data size (64, 1024) for conversion as an example, *tensor_size* is 65536 (i.e., 64 × 1024).

$$casting\_cost = r * tensor\_size + C_I \quad (1)$$

To use the above model, we must know r and C_I. They are obtained using linear regression where r and C_I are the slop coefficient and intercept. In particular, by profiling 500 FP32-to-FP16 cast cases and 500 FP16-to-FP32 cast cases, we collect a total of 1000 training samples, each of which includes a pair of measured *casting_cost* and *tensor_size* from a cast operation. We use the method of least squares to find the values of r and C_I that minimize the sum of the squared errors.

4.2.2 Predicting Execution Time of Operation

We adopt an operation-specific modeling method, which means that we build a performance model for each individual operation. We do not build a general model for all the operations to predict performance, because the operations exhibit a variety of performance characteristics in terms of memory access patterns and computation intensity. Building a single, general model does not give good prediction accuracy. In our experience, we build a general model using the similar method as building operation-specific models, but can achieve only 43% prediction accuracy, which leads to 21% longer execution time, compared with using operation-specific models. Furthermore, although the total number of operations in the *allowlist*, *inferlist*, and *clearlist* is 143, which is large, the operation-specific performance models are built offline for once, and then can be reused for all NN models. Hence, operation-specific modeling is practical.

Why not using dynamic profiling to measure performance in low precision? Our performance modeling predicts

the execution time of an operation using low precision on regular CUDA cores and TC (if the operation is an TC candidate). We could use dynamic profiling to measure the execution time. In particular, we use three training iterations of the NN model: one iteration running all operation nodes in FP32, one running all operation nodes in FP16 on regular CUDA cores, and one iteration running TC candidate nodes in FP16 on TC.

However, the dynamic profiling using training iterations has limitations. In particular, the dynamic profiling uses FP16 for many operations to measure execution time, regardless of the impact of those operations on training convergence. To avoid slow convergence of the training process after dynamic profiling, one has to discard the three training iterations and restart the training process. This complicates the training pipeline. Even worse, for those downstream training tasks that use a pre-trained model, the number of training iterations is limited. Using dynamic profiling can lead to a large increase in training time. For example, training (fine-tuning) a small NN model Audio2Vec [15] can take about ten iterations, and losing three iterations for dynamic profiling leads to more than 20% increase in training time.

Modeling intuition. The performance of an operation in FP32 and FP16 has correlation. For example, using FP16 reduces the working set size, compared with using FP32, which can lead to less cache misses and in turn decrease execution time. Our intuition is that using execution time in FP32 and a handful of performance-critical events measured in FP32, we can predict execution time in FP16. The parameters (coefficients) in our performance modeling should capture the performance correlation between FP32 and FP16.

Performance modeling. Based on the above intuition, we build the performance model as follows.

$$OP_{LP} = OP_{FP32} \cdot (\sum_{i=1}^{N} w_i \cdot PC_i) + \sigma \quad (2)$$

where OP_{LP} is the execution time on either regular CUDA cores or TC using FP16, OP_{FP32} is the execution time on regular CUDA cores using FP32, PC_i is a performance-critical event measured during the execution of FP32, N is the number of performance critical events, and w_i and σ are coefficients. Each operation has up to two performance models: one for regular CUDA cores and the other for TC.

A performance-critical event is a model feature. We use hardware performance counters on GPU to collect those events. There are about 100-200 events collectable on GPU. We choose those events that are the most correlated to the operation performance in FP16, using the following method.

Selection of model features. We use the Spearman's rank correlation coefficient [16] (or Spearman's ρ) to select events. The Spearman's ρ is a method to quantify how well the relationship between two variables can be described using a monotonic function [16]. The Spearman correlation between two variables is high when observations have a similar rank between the two variables, and low when observations have a dissimilar rank between the two variables.

In our case, an event is a variable and the operation performance using FP16 is the other variable. We make many observations by running tests. If the observations on the event and the observations on the F16 performance have the similar rank (i.e., relative position label of the observations within the event or F16 performance), then the event is monotonically correlated (either monotonically increasing or monotonically decreasing) with the FP16 performance. In our case, we use a threshold of 0.75 for ρ. When $|\rho|$ for an event is larger than 0.75, then we choose it as a model feature.

In particular, given an operation `OP`, we use the following method to select features to build the performance model for FP16 on TC. Using 1000 different TC-satisfied inputs, we run `OP` in FP32 1000 times to collect execution time and each collectable event. Then, using the same 1000 inputs, we run `OP` in FP16 1000 times to collect execution time. As such, we construct 1000 samples, each of which consists of measured FP32 execution time, the value of each collectable event, and FP16 execution time. For each event, we use the 1000 samples to run the Spearman's rank correlation analysis and calculate ρ. If ρ is larger than the threshold, then the event is selected.

To select the features to build the performance model for FP16 on regular CUDA cores, we use the same method as above but the operation inputs do not necessarily meet the TC requirement.

We select events for each operation using the above approach. We discuss the most common events across operations as follows.

- **Global memory load/store throughput** indicates intensiveness of global memory access. If an operation has higher throughput in global memory load/store, the operation may get larger performance benefit by using FP16.
- **Instruction executed per cycle** indicates compute intensity. Using FP16 can be helpful for those floating point intensive operations, because of higher throughput of FP16 instructions.
- **GPU occupancy** indicates how many warps are able to be active during operation execution. An operation with low GPU occupancy will be sensitive to whether FP16 or FP32 is used, because using FP16 or not causes difference in global memory accesses and the operation with low GPU occupancy has lower thread-level parallelism to hide long global-memory access latency.
- **L2 cache accesses and misses and L1 cache accesses** indicate memory access locality in the operation. Compared with an operation with bad reference locality, an operation with good reference locality can take advantage of the cache hierarchy and is not sensitive to global memory bandwidth, hence less sensitive to the global memory bandwidth savings due to the use of FP16.

Getting model coefficients w_i and σ. For each operation-specific performance model, we run the operation 1000 times

with 1000 inputs with different sizes, using FP32 and FP16 respectively. That generates 1000 samples, each of which includes FP32 execution time, the values of events collected in FP32, and FP16 execution time. Using the 1000 samples, we use the method of least squares to find the values of coefficients w_i and σ that minimize the sum of the squared errors. To generate 1000 inputs with different sizes, we profile the input data size of the operations from 11 NN models (including GoogLeNet, UNet-3D, DLRM, DCIGN, BiLSTM, SSD-MobileNet-v1, ShuffleNet, SSD, DenseNet, Mask R-CNN, RNN-T) from the MLPerf benchmark suite [17] using 100 training steps with various batch sizes.

Building performance models for an operation is not time-consuming. For example, building the two models for the operation `Conv2D` (including generating samples to get the model coefficients) takes about 1.5 hours. Building performance models for 143 operations takes about 112.5 hours.

Justification of modeling method. In essence, our modeling method is linear regression. Before we used it, we asked if other modeling methods (such as using a machine learning model) can work. We built a multilayer perceptron model (MLP) taking the same input and output as our linear regression model. The MLP has four layers (one input, one output and two hidden layers) and has 800 neurons in total. However, we do not see any benefit of using such a model in terms of prediction accuracy: the prediction accuracy for the MLP is 71%, while it is 94.2% for the linear regression model. The low prediction accuracy of using MLP is largely due to the fact that our problem nature exhibits near-linear correlation between features and MLP seems to be prone to get stuck in a local optimum for this problem. Furthermore, the MLP takes 10x more training samples than the linear regression model. Hence, we do not use MLP.

Furthermore, we asked if basic heuristics can work. In fact, compared with using dynamic profiling (a basic heuristic), using performance modeling reduces training time by 20% for Audio2Vec. Compared with using the same precision for all instances of each operation (another basic heuristic), using performance modeling reduces training time by 35% for BERT-large. Our performance models are repeatedly used for NN models, which amortizes the construction cost.

Our modeling method considers the impact of operation input on operation performance, because it uses dynamic profiling in FP32 to measure performance with the given operation input, based on which to make prediction for FP16 performance with the same operation input.

Furthermore, our modeling method is operation-specific, which greatly simplifies model construction, because the operation type itself provides much of implicit information on operation characteristics. For example, the operation name `MatMul` indicates strided memory accesses, and hence the operation-specific performance model does not need to explicitly model such a memory access pattern to make performance prediction. As a result, our performance model can focus on capturing the correlation between the performance of FP32 and FP16.

4.3 Runtime Graph Rewriting

The runtime graph rewriting decides (1) data precision for each operation, and (2) which operations to be converted together to reduce the number of cast operation nodes. By graph rewriting, Campo aims to reach the following goals: (1) minimizing the training time; (2) minimizing the casting cost; and (3) no adverse impact on the numerical safety (compared with the traditional mixed-precision training).

The graph rewriting in Campo includes graph profiling, graph traverse to determine the precision assignments, and insertion of cast operation nodes, discussed as follows.

Graph profiling. Given an NN model, Campo uses a single training step (or iteration) running in FP32 to collect execution time and events needed by performance modeling for those operations in *allowlist*, *inferlist*, and *clearlist*. The graph profiling is triggered right after the first few training steps used by TensorFlow for warmup (i.e., determining system configurations).

Graph traverse happens after graph profiling, and performs four times on the dataflow graph. Each graph traverse follows the data flow in the NN training to analyze operation nodes in the dataflow graph. During the graph traverses, Campo uses two lists, *allow_nodes* and *deny_nodes*, to record those operation nodes determined to run in FP16 and in FP32 respectively. We depict the four-time traverse as follows.

Traverse #1. During the traverse, when an operation node is encountered, Campo checks if it is in *allowlist*. If yes, then Campo uses performance modeling to decide if the casting cost plus FP16 execution time of the operation (on regular CUDA cores or TC) is smaller than the counterpart FP32 execution time. If yes, then the operation node is put into *allow_nodes*; If no, then it is put into *deny_nodes*. During the traverse, only those operations that are numerical safe in FP16 (i.e., in *allowlist*) are considered, in order to maintain the training accuracy of the NN model.

Traverse #2. During this traverse, Campo checks the remaining operation nodes. For each node, Campo checks if it is either numerically-unsafe (i.e., in *denylist*) or on a path from a node in *denylist* to another node in *denylist* or *inferlist* through some operation nodes in *inferlist* or *clearlist*. If yes, then the checking node is added to *deny_nodes*. This traverse aims to prevent numerically-unsafe operation nodes and their downstream operation nodes from being changed to FP16, in order to maintain the training accuracy.

Traverse #3. During this traverse, Campo checks each remaining operation node. If the node (called the target node in the remaining discussion) is in *inferlist* or *clearlist*, then Campo put the target node into *allow_nodes*. After Traverse 2, such a node should be safe to use FP16. It is possible that the immediate upstream or downstream node(s) of the target

node is in *allow_nodes*. For such a case, the cast operation to convert target node for FP16 or FP32 is saved for higher performance.

Traverse #4. During this traverse, Campo checks each remaining operation node. If the node (called the target node in the remaining discussion) is in *clearlist* and connected to a node in the *allow_nodes* via other nodes in *clearlist*, then Campo uses performance modeling to decide whether the casting cost is smaller than the performance benefit of using FP16 (on regular CUDA cores or TC) for the target node and other connecting nodes. If yes, then the target node is put into *allow_nodes*.

Insertion of cast operation nodes. After the four-time graph traverse, Campo changes the type attribute of operation nodes according to their FP16 or F32 assignments, and then inserts a cast operation node at the boundary between any FP16 node and its neighbour FP32 node (or vice versa) by using the API `ChangeTypeAttrsAndAddCasts` provided by TensorFlow.

4.4 Usage of Tensor Cores

For any operation node in FP16 decided in the graph rewriting process, Campo is able to use performance modeling to decide if using regular CUDA cores or TC is more performance beneficial. If using TC is better, then Campo makes the best efforts to run the operation on TC.

In particular, in each training step, Campo checks the input shape of each TC candidate. If the input shape of a TC candidate cannot meet the TC requirements, Campo pads the input tensor by adding zero-filled rows or columns. For example, for `Conv2D`, Campo pads its input to make the dimension size of each channel a multiple of 8. Compared to the traditional padding method recommended by dlprof, our method is implemented inside the training framework, and thus transparent to the users and does not need to modify NN models.

Overhead analysis. Zero padding adds overhead to computation and memory consumption. For an operation decided to use FP16 on TC by performance modeling, the computation overhead (with casing cost) is easily surpassed by the performance benefit: in our evaluation, an operation decided to use FP16 on TC by performance modeling can typically gain about 2x performance improvement (compared with using FP32 on regular CUDA cores), while padding usually leads to less than 20% performance overhead. To consider the performance overhead of zero padding in performance modeling, we can introduce a threshold, which is an empirical estimation on the padding overhead. Only when the casting cost plus this threshold is smaller than performance benefit, we use FP16 on TC.

The memory overhead of padding is usually less than 1%, which is very small. This is because in practice, the number of zero-filled rows or columns via padding is less than 8 and the number of dimensions requiring padding is typically at most 2, while the total number of rows or columns is hundreds.

5 Implementation

Campo extends the mixed precision graph optimizer and runtime system in TensorFlow v1.15. Such an extension includes 235 C++ LOC. The extension is used to decide if an operation should use FP16 based on performance modeling. We add APIs `CheckAllowListOps` and `CheckIfAllowThroughClear` to implement the first and fourth graph traverses. The other two traverses extend the existing implementation for assigning precision to operations in TensorFlow. In TensorFlow's `op_kernel` module, we add an API `InputShapeTranform` to implement input padding and meet the TC requirements on the input shape. Besides the above extension, Campo includes graph profiling and performance modeling (including offline tools to build performance models). In total, Campo is written in 2570 LOC.

6 Discussions

Differences between using performance modeling and static profiling. The static profiling is an alternative approach to get performance of operations in low precision. Using static profiling, the user must collect the information on tensor shapes from operators, and then use the collected information to run operators in low precision offline. We discuss the differences between performance modeling and static profiling as follows.

There are two differences. First, the static profiling has to be done for each NN model and is not scalable, while the performance modeling, once built, can generally work for most NN models. Second, when the number of tensor shapes and operations in an NN model for profiling is small, the static profiling is a better solution to get operation performance in low precision. However, the profiling cost must be small enough to enable practical deployment of static profiling. In contrast, the performance modeling does not incur deployment cost for most of NN models. The performance models can be repeatedly used for NN models, which amortizes the model construction cost.

Portable performance modeling. Our performance modeling is architecture-dependent, because it collects architecture-dependent performance events as the model features. This means that we must build different performance models for different GPU architectures. How to reduce human efforts to build performance models remains to be studied. In addition, it would be interesting to extend Campo to lower precisions (e.g., INT8 and BF16) using the same methodology in Campo. We leave them as our future work.

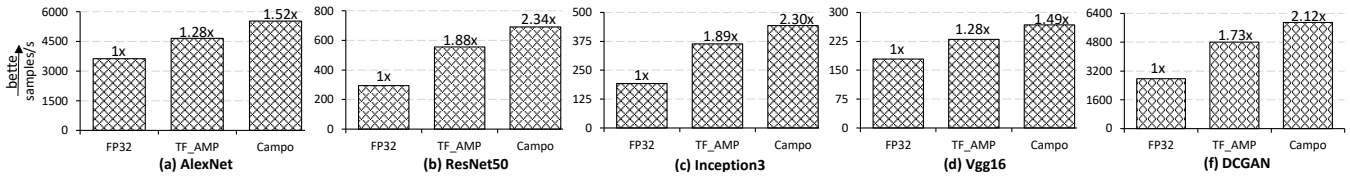

Figure 4: Training throughput with FP32, TF_AMP and Campo on RTX 2080 Ti.

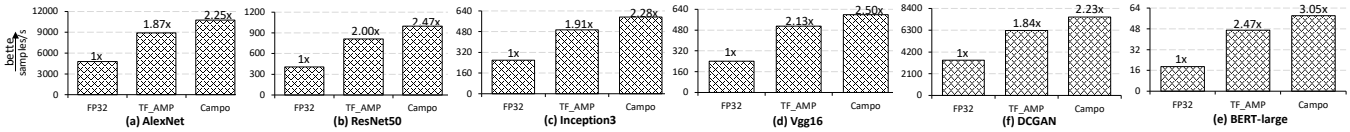

Figure 5: Training throughput with FP32, TF_AMP and Campo on V100.

Table 2: Hardware configurations

CPU	Intel Xeon CPU E5-2648L v4@ 1.80GHz
Main Memory	64 GB DDR4
CPU Cores	2 sockets, 14 cores per socket
GPU	NVIDIA GeForce RTX 2080 Ti (Turing)
CUDA Cores	4352 CUDA cores (68 SMs, 1.54GHz)
Tensor Cores	544 tensor cores
L1 Cache	64 KB (per SM)
L2 Cache	5.767 MB
GPU Device Memory	11 GB GDDR6
GPU	NVIDIA Tesla V100 (Volta)
CUDA Cores	5376 CUDA cores (84 SMs, 1.53GHz)
Tensor Cores	672 tensor cores
L1 Cache	128 KB (per SM)
L2 Cache	6.144 MB
GPU Device Memory	32 GB HBM2

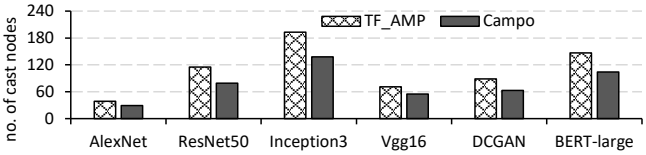

Figure 6: The number of cast nodes of NN models trained with TF_AMP and Campo, respectively.

7 Evaluation

7.1 Experimental Setup

Experimental Platforms and Tools. We use a multicore machine equipped with two TC-supported GPUs (i.e., Nvidia GeForce RTX 2080 Ti and V100) and Intel Xeon CPU listed in Table 2. The two GPUs are attached to the server by PCIe 3.0. We use CUDA 9.0 [18], NVIDIA cuDNN 8.0, and Ubuntu 18.04. We use dlprof [14] and Nvidia Nsight Compute [19] to collect performance statistics. We measure system power for GPU, CPU and DRAM by using a collection of industry-standard tools including NVIDIA System Management Interface [20] and Intel Running Average Power Limit (RAPL) Interface [21]. We use the number of samples processed per second and training throughput per Watt as metrics to quantify training throughput and energy efficiency respectively. Unless indicated otherwise, the reported results are collected on V100 and all tests use the default GPU setting.

Benchmarking Methodology. We evaluate six NN models including AlexNet [22], Inception3 [23], Vgg16 [24], ResNet50 [25], DCGAN [26], and BERT-large [6]. We only evaluate BERT-large on V100 because of the out-of-memory error on RTX 2080 Ti. For the first four models, Imagenet is used as the training dataset [27]. For DCGAN and BERT-large, we use CelebA [28] and SQuAD [29] as the training dataset, respectively. The training batch size for AlexNet and other models on GeForce RTX 2080 Ti GPU is 256 and 64 respectively, according to the model configurations in related work [30]. The training batch size for BERT-large and other models on V100 is 10 and 256 respectively, according to the model configurations in related work [31].

We run each NN model training experiment ten times and then report the average results. We use TensorFlow v1.15 and its performance optimizer "TF_AMP" for mixed precision training. TF_AMP is our baseline for performance comparison. TF_AMP in TensorFlow v1.15 is the state-of-the-art solution and *the most recent* performance optimizer for mixed precision training. To preserve small gradient values, we adopt the automatic loss-scale optimizer in TF_AMP. To evaluate model accuracy, We test both the Top-1 and Top-5 accuracy on the ImageNet-1k validation set for AlexNet, ResNet50, Inception3 and Vgg16, and the celebA validation set for DCGAN. For BERT, we test the F1 score on the SQuAD v1.1 validation set.

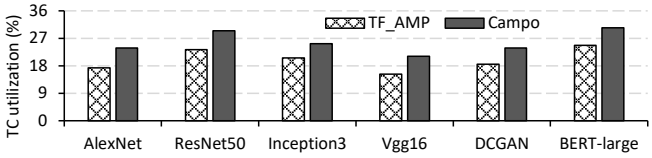

Figure 7: TC utilization of NN models trained with TF_AMP and Campo, respectively.

7.2 Training Throughput

Figures 4 and 5 show the training throughput of Campo, TF_AMP, and single precision training (using FP32). To calculate speedup, the throughput of using the single precision training is used as the baseline.

Using mixed precision training, TF_AMP and Campo achieve large speedup (1.28x - 3.05x) over FP32 on both GPUs. Furthermore, Campo outperforms TF_AMP by 20.8% on average (up to 24.5%) on RTX 2080 Ti, as well as by 20.9% on average (up to 23.4%) on V100. The performance benefit of Campo over TF_AMP comes from two perspectives: reducing the number of cast operation nodes in the dataflow graph and using TC more often, discussed as follows.

7.3 Performance Breakdown

Number of cast operation nodes. Figure 6 quantifies the number of cast operation nodes. Campo reduces the number of cast operations nodes for higher performance. Campo uses 27.7% less cast operation nodes on average (up to 31.3%) than TF_AMP.

TC utilization is defined as the percentage of training time when TC is busy. Figure 7 shows TC utilization. Campo increases the utilization of TC by 29.4% on average (up to 37.9%), which indicates that Campo uses TC more often.

Contribution quantification of the graph rewriting and improving TC utilization. We disable our method of improving TC utilization but keep the graph rewriting to quantify its contribution to the performance improvement (compared with TF_AMP). Then we enable our method of improving TC utilization along with the graph rewriting to quantify the contribution of improving TC utilization. Figure 8(a) and Figure 8(b) show the results on two GPUs. The two figures reveal that the graph rewriting contributes more than improving TC utilization: 84.5% of the overall performance improvement (on average) comes from the graph rewriting.

We also notice that V100 benefits 38.1% more (on average) from TC, compared with RTX 2080 Ti. This is because of two reasons. (1) V100 has more computation resource: V100 has 23.5% more tensor cores than RTX 2080 Ti; (2) training on V100 is able to run 10.4% more operations on TC than training on RTX 2080 Ti, because higher performance benefits of using TC on V100 offset casting cost in more operations.

Table 3: Model accuracy of NN models trained with FP32, TF_AMP and Campo, respectively.

NN models	Top-1 Accuracy (%)			Top-5 Accuracy (%)		
	FP32	TF_AMP	Campo	FP32	TF_AMP	Campo
AlexNet	63.39	64.41	64.38	81.24	81.21	81.19
ResNet50	78.77	78.74	78.75	94.86	94.82	94.85
Inception3	78.42	78.45	78.43	90.15	90.16	90.15
Vgg16	71.58	71.6	71.57	88.28	88.25	88.27
DCGAN	80.12	80.16	80.13	92.47	92.46	92.44
BERT-large	91.35	91.36	91.33	N/A		

7.4 Training Accuracy

Table 3 reports the model accuracy of six NN models trained with FP32, TF_AMP and Campo, respectively. We can see that across NN models, the model training with Campo leads to no loss in model accuracy compared to TF_AMP, which closely matches the FP32 training accuracy (the subtle differences across FP32, TF_AMP and Campo in accuracy are within typical bounds of run-to-run variations).

Campo preserves training accuracy, because of two reasons. (1) Those operations that are numerical unsafe still use FP32. (2) Campo uses the same effective loss-scaling optimizer as TF_AMP to preserve small gradients in FP16.

7.5 Prediction Accuracy of Performance Models

To evaluate the accuracy of the performance models, we use a metric denoted by M_A as follows.

$$M\_A = 1 - \frac{1}{n}\sum_{i=1}^{n}\left|\frac{\hat{y}_i - y_i}{y_i}\right| \quad (3)$$

where n is the number of test cases, and $\hat{y}_i$ and y_i are the predicted and measured execution time for the test case i.

We test 150 performance models for 143 operations respectively. For each model, we use 100 different input sizes as test cases. In total, there are 15000 tests. We report the modeling accuracy for five common operations, i.e., `Cast`, `MatMul`, `Conv2DBackpropFilter`, `Conv2DBackpropInput` and `Conv2D` in Figure 9.

In general, the average prediction errors for the five operations are less than 5%, which demonstrates high prediction accuracy. Overall, the prediction error for 143 operations is 5.8% on average (and less than 6%).

Handling mis-prediction of performance modeling. When a mis-prediction happens, there are two possible outcomes. (1) Based on performance modeling, the operation is not scheduled to run in FP16, although it should be for better performance. (2) Based on performance modeling, the operation is scheduled to run in FP16, although it should not, because of high casting cost.

For the case (1), mis-prediction does not cause any performance loss, compared with using full precision (i.e., the

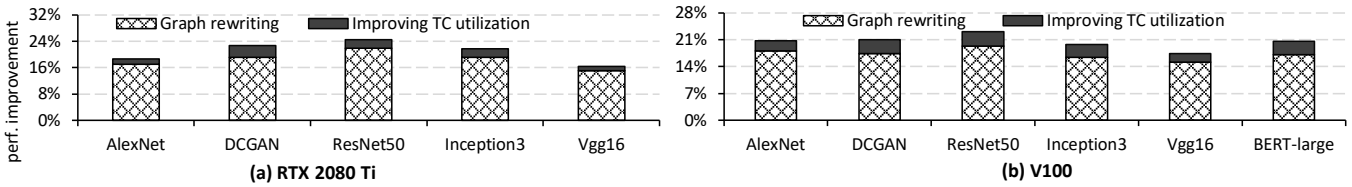

Figure 8: Breakdown of the overall performance improvement from graph rewriting and improving TC utilization.

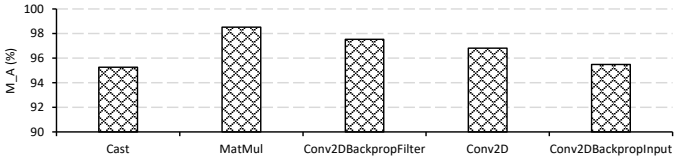

Figure 9: Performance prediction accuracy for five operations based on the operation-specific performance models

Table 4: Average system power consumption of NN models trained with FP32, TF_AMP and Campo on RTX 2080 Ti and V100, respectively.

NN Models	Average System Power (W)					
	RTX 2080 Ti			V100		
	FP32	TF_AMP	Campo	FP32	TF_AMP	Campo
AlexNet	274	268	267	325	319	316
ResNet50	272	265	263	324	313	311
Inception3	273	264	263	326	316	315
Vgg16	273	267	267	324	316	316
DCGAN	275	268	267	327	320	319
BERT-large	N/A	N/A	N/A	332	320	318

original execution). For the case (2), mis-prediction causes performance loss in that operation. But since the performance prediction accuracy is high, the performance loss is easily outweighed by the performance benefit of correctly using FP16 in other operations. In our evaluation, for each NN model, the case (2) happens at most 7 times, taking less than 1.5% of all prediction cases.

No matter whether the case (1) or (2) happens, neither of them causes any loss in training accuracy of the NN model, because the performance modeling is never applied to any numerical-unsafe operation and hence the mis-prediction never happens to any of them.

7.6 Power Consumption and Energy Efficiency

Power consumption. Table 4 summarizes system power consumption. Using TF_AMP and Campo, the system consumes less power than using FP32 across NN models by 6 - 13 Watts, because of the use of power-efficient TC in mixed precision training. Table 4 also shows that using Campo, the system consumes less power than using TF_AMP, due to better utilization of TC in Campo.

Energy efficiency. Figure 10 shows the energy efficiency of NN models trained with FP32, TF_AMP and Campo. TF_AMP and Campo outperform FP32 by 109.5% and 154.6% on average, respectively, because of the use of reduced precision and power efficient TC. Compared to TF_AMP, Campo outperforms TF_AMP by 21.4% on average (up to 24.2%). This improvement comes from the fact that Campo leads to better performance (see Figure 5) without causing extra power consumption (see Table 4).

8 Related Work

Mixed precision for NN training. Many research efforts have been dedicated to achieve more efficient NN training with mixed precision. Mixed precision training was first introduced by Micikevicius et al [1]. Since then, Nvidia depicts how to use it with TC [3]. Jia et al. [2] use mixed precision to improve scalability of synchronized stochastic gradient descent (SGD) optimizers in NN models without losing model generability. Kuchaiev et al. [32] presents a TensorFlow-based toolkit for mixed precision training of sequence-to-sequence models, with an implementation of a wrapper around the standard TensorFlow performance optimization facility for mixed precision training. Besides the floating-point based mixed precision training, Das et al. [4] are the first to propose mixed precision training of convolutional neural networks using integer operations on ImageNet-1K dataset. Svyatkovskiy et al. [33] introduce a learning rate schedule for training distributed deep recurrent neural networks with mixed precision on GPU clusters. Their schedule facilitates neural network convergence at up to O(100) workers.

Different from the above efforts, our work reveals the overlooked performance issues related to casting cost in mixed precision graph optimization.

Mixed precision for other GPU applications. Mixed precision has been explored to speed up dense linear system solvers [34, 35] and WZ factorization [36] in the context of HPC. Kotipalli et al. [37] present AMPT-GA, an automatic mixed precision optimization system that automatically selects the optimal data precision to maximize performance while meeting accuracy constraints for GPU Applications. However, its evaluation is only limited to an NVIDIA Tesla P100 GPU machine without TC. Haidar et al. [38] apply mixed-precision FP16–FP32/FP64 to a high-performance iterative refinement solvers and take advantage of TC. Gallo et al. [39] propose the concept of mixed-precision in-memory

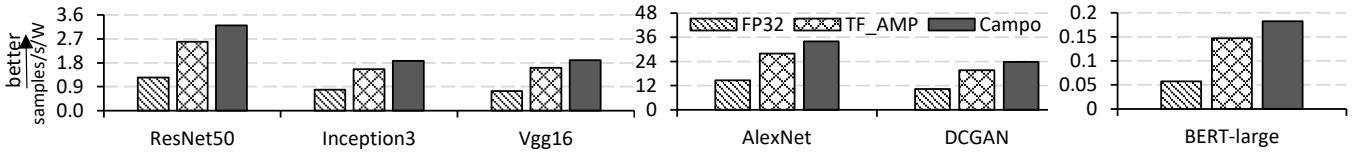

Figure 10: Energy efficiency of NN models trained with FP32, TF_AMP and Campo.

computing with a combination of a von Neumann machine and a computational memory unit. Baboulin et al. [40] leverage mixed precision to accelerate computations in many dense and sparse linear algebra algorithms. Lam et al. [41] introduce a framework that employs binary instrumentation and modification to build mixed-precision configurations of existing binaries originally developed for the use of double precision. In contrast, our work focuses on applying mixed precision to NN training.

Investigation on the usage of TC. Nvidia TC follows the IEEE 754 standard [42] and utilizes mixed precision with matrix multiplication input in FP16 and accumulation in FP32. Motivated by the benefits of TC, Yan et al. [43] demystify how TC on Turing GPUs works and implement TC-based HGEMM on NVIDIA Turing GPUs. Markidis et al. [44] perform a performance evaluation of TC in mixed precision as well as three different approaches of programming matrix-multiply-and-accumulate on TC on V100. Brennan et al. [45] perform a thorough analysis of the effects of low precision operations and TC on graph convolutional neural networks. Abdelfattah et al. [46] explore leveraging TC to implement a optimized batched Matrix multiplication (HGEMM) in half-precision arithmetic. Our work is different from these efforts, as we explore the use of TC in NN training.

9 Conclusions

This paper introduces Campo, a cost-aware performance optimization tool for mixed-precision NN training that assigns the optimal precision (either FP32 or FP16) to training operations while minimizing the unnecessary casts to maximize the training performance. Campo is based on our unique observation that the casting cost to achieve mixed precision training may offset the performance benefit of using low precision in mixed precision training. This observation is ignored in the existing approaches of using mixed precision, which leads to smaller performance improvement or even performance loss. We build operation-specific performance models to predict and quantify the impact of casting cost on the performance of using low precision. With the performance models, at runtime Campo employs a cost-aware graph rewriting strategy to make decisions on which precision should be used for each operation without losing NN training accuracy. We evaluate Campo with six NN models on Nvidia Turing and Volta architecture-based GPUs, and show that Campo largely outperforms TensorFlow.

10 Acknowledgements

We thank anonymous reviewers and our shepherd for their valuable feedback. This work is partially supported by the National Science Foundation of China under grants 61972137 and 61772183.

References

[1] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, et al. Mixed precision training. *arXiv preprint arXiv:1710.03740*, 2017.

[2] Xianyan Jia, Shutao Song, Wei He, Yangzihao Wang, Haidong Rong, Feihu Zhou, Liqiang Xie, Zhenyu Guo, Yuanzhou Yang, Liwei Yu, et al. Highly scalable deep learning training system with mixed-precision: Training imagenet in four minutes. *arXiv preprint arXiv:1807.11205*, 2018.

[3] Nvidia. Nvida's mixed-precision training - tensorflow example. https://docs.nvidia.com/deeplearning/performance/mixed-precision-training, 2018.

[4] Dipankar Das, Naveen Mellempudi, Dheevatsa Mudigere, Dhiraj Kalamkar, Sasikanth Avancha, Kunal Banerjee, Srinivas Sridharan, Karthik Vaidyanathan, Bharat Kaul, Evangelos Georganas, et al. Mixed precision training of convolutional neural networks using integer operations. *arXiv preprint arXiv:1802.00930*, 2018.

[5] Google. Tensorflow - enable mixed precision graph rewrite. https://www.tensorflow.org/versions/r1.15/api_docs/python/tf/train/experimental/enable_mixed_precision_graph_rewrite, 2018.

[6] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[7] Muthian Sivathanu, Tapan Chugh, Sanjay S Singapuram, and Lidong Zhou. Astra: Exploiting predictability to

optimize deep learning. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 909–923, 2019.

[8] Mark Hildebrand, Jawad Khan, Sanjeev Trika, Jason Lowe-Power, and Venkatesh Akella. Autotm: Automatic tensor movement in heterogeneous memory systems using integer linear programming. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 875–890, 2020.

[9] Xufan Zhang, Ziyue Yin, Yang Feng, Qingkai Shi, Jia Liu, and Zhenyu Chen. Neuralvis: Visualizing and interpreting deep learning models. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1106–1109. IEEE, 2019.

[10] Jiawen Liu, Dong Li, Gokcen Kestor, and Jeffrey Vetter. Runtime concurrency control and operation scheduling for high performance neural network training. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 188–199. IEEE, 2019.

[11] Xin He, Jiawen Liu, Zhen Xie, Hao Chen, Guoyang Chen, Weifeng Zhang, and Dong Li. Enabling energy-efficient dnn training on hybrid gpu-fpga accelerators. In *Proceedings of the ACM International Conference on Supercomputing*, pages 227–241, 2021.

[12] Jiawen Liu, Hengyu Zhao, Matheus A Ogleari, Dong Li, and Jishen Zhao. Processing-in-memory for energy-efficient neural network training: A heterogeneous approach. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 655–668. IEEE, 2018.

[13] Xuan Peng, Xuanhua Shi, Hulin Dai, Hai Jin, Weiliang Ma, Qian Xiong, Fan Yang, and Xuehai Qian. Capuchin: Tensor-based gpu memory management for deep learning. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 891–905, 2020.

[14] Dlprof - nvidia deep learning frameworks documentation. https://docs.nvidia.com/deeplearning/frameworks/dlprof-user-guide/, 2021.

[15] Marco Tagliasacchi, Beat Gfeller, Félix de Chaumont Quitry, and Dominik Roblek. Self-supervised audio representation learning for mobile devices. *arXiv preprint arXiv:1905.11796*, 2019.

[16] Thomas W MacFarland and Jan M Yates. Spearman's rank-difference coefficient of correlation. In *Introduction to nonparametric statistics for the biological sciences using R*, pages 249–297. Springer, 2016.

[17] Peter Mattson, Christine Cheng, Cody Coleman, Greg Diamos, Paulius Micikevicius, David Patterson, Hanlin Tang, Gu-Yeon Wei, Peter Bailis, Victor Bittorf, et al. Mlperf training benchmark. *arXiv preprint arXiv:1910.01500*, 2019.

[18] Cuda toolkit documentation v9.0. https://developer.nvidia.com/cuda-toolkit-archive, 2019.

[19] Nvidia nsight compute - nvidia developer documentation. https://developer.nvidia.com/nsight-compute, 2021.

[20] Nvidia. Nvidia system management interface. https://developer.nvidia.com/nvidia-system-management-interface, 2018.

[21] Intel's runningaverage power limit (rapl) interface. https://01.org/rapl-power-meter, 2019.

[22] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[23] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2818–2826, 2016.

[24] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[25] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[26] Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv preprint arXiv:1511.06434*, 2015.

[27] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. IEEE, 2009.

[28] Ziwei Liu, Ping Luo, Xiaogang Wang, and Xiaoou Tang. Deep learning face attributes in the wild. In *Proceedings of International Conference on Computer Vision (ICCV)*, December 2015.

[29] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. SQuAD: 100,000+ questions for machine comprehension of text. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pages 2383–2392, Austin, Texas, November 2016. Association for Computational Linguistics.

[30] Chuan Li. Rtx 2080 ti deep learning benchmarks with tensorflow. https://lambdalabs.com/blog/2080-ti-deep-learning-benchmarks/, 2019.

[31] Nvidia data center deep learning product performance. https://developer.nvidia.com/deep-learning-performance-training-inference, 2020.

[32] Oleksii Kuchaiev, Boris Ginsburg, Igor Gitman, Vitaly Lavrukhin, Jason Li, Huyen Nguyen, Carl Case, and Paulius Micikevicius. Mixed-precision training for nlp and speech recognition with openseq2seq. *arXiv preprint arXiv:1805.10387*, 2018.

[33] Alexey Svyatkovskiy, Julian Kates-Harbeck, and William Tang. Training distributed deep recurrent neural networks with mixed precision on gpu clusters. In *Proceedings of the Machine Learning on HPC Environments*, pages 1–8. 2017.

[34] Alfredo Buttari, Jack Dongarra, Julie Langou, Julien Langou, Piotr Luszczek, and Jakub Kurzak. Mixed precision iterative refinement techniques for the solution of dense linear systems. *The International Journal of High Performance Computing Applications*, 21(4):457–466, 2007.

[35] Azzam Haidar, Panruo Wu, Stanimire Tomov, and Jack Dongarra. Investigating half precision arithmetic to accelerate dense linear system solvers. In *Proceedings of the 8th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*, pages 1–8, 2017.

[36] Beata Bylina and Jarosław Bylina. Mixed precision iterative refinement techniques for the wz factorization. In *2013 Federated Conference on Computer Science and Information Systems*, pages 425–431. IEEE, 2013.

[37] Pradeep V Kotipalli, Ranvijay Singh, Paul Wood, Ignacio Laguna, and Saurabh Bagchi. Ampt-ga: automatic mixed precision floating point tuning for gpu applications. In *Proceedings of the ACM International Conference on Supercomputing*, pages 160–170, 2019.

[38] Azzam Haidar, Stanimire Tomov, Jack Dongarra, and Nicholas J Higham. Harnessing gpu tensor cores for fast fp16 arithmetic to speed up mixed-precision iterative refinement solvers. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 603–613. IEEE, 2018.

[39] Manuel Le Gallo, Abu Sebastian, Roland Mathis, Matteo Manica, Heiner Giefers, Tomas Tuma, Costas Bekas, Alessandro Curioni, and Evangelos Eleftheriou. Mixed-precision in-memory computing. *Nature Electronics*, 1(4):246–253, 2018.

[40] Marc Baboulin, Alfredo Buttari, Jack Dongarra, Jakub Kurzak, Julie Langou, Julien Langou, Piotr Luszczek, and Stanimire Tomov. Accelerating scientific computations with mixed precision algorithms. *Computer Physics Communications*, 180(12):2526–2533, 2009.

[41] Michael O Lam, Jeffrey K Hollingsworth, Bronis R de Supinski, and Matthew P LeGendre. Automatically adapting programs for mixed-precision floating-point computation. In *Proceedings of the 27th international ACM conference on International conference on supercomputing*, pages 369–378, 2013.

[42] Nathan Whitehead and Alex Fit-Florea. Precision & performance: Floating point and ieee 754 compliance for nvidia gpus. *rn (A+ B)*, 21(1):18749–19424, 2011.

[43] Da Yan, Wei Wang, and Xiaowen Chu. Demystifying tensor cores to optimize half-precision matrix multiply. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 634–643. IEEE, 2020.

[44] Stefano Markidis, Steven Wei Der Chien, Erwin Laure, Ivy Bo Peng, and Jeffrey S Vetter. Nvidia tensor core programmability, performance & precision. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 522–531. IEEE, 2018.

[45] John Brennan, Stephen Bonner, Amir Atapour-Abarghouei, Philip T Jackson, Boguslaw Obara, and Andrew Stephen McGough. Not half bad: Exploring half-precision in graph convolutional neural networks. In *2020 IEEE International Conference on Big Data (Big Data)*, pages 2725–2734. IEEE, 2020.

[46] Ahmad Abdelfattah, Stanimire Tomov, and Jack Dongarra. Fast batched matrix multiplication for small sizes using half-precision arithmetic on gpus. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 111–122. IEEE, 2019.