CCF Transactions on High Performance Computing
https://doi.org/10.1007/s42514-024-00208-9

REGULAR PAPER

Convergence-aware operator-wise mixed-precision training

Wenhao Dai[1] · **Ziyi Jia**[1] · **Yuesi Bai**[1] · **Qingxiao Sun**[1]

Received: 29 July 2024 / Accepted: 10 November 2024

Abstract

With the support of more precision formats in emerging hardware architectures, mixed-precision has become a popular approach to accelerate deep learning (DL) training. Applying low-precision formats such as FP16 and BF16 to neural operators can save GPU memory while improving bandwidth. However, DL frameworks use black and white lists as default mixed-precision selections and cannot flexibly adapt to a variety of neural networks. In addition, existing work on automatic precision adjustment does not consider model convergence, and the decision cost of precision selection is high. To address the above problems, this paper proposes *CoMP*, a non-intrusive framework for Convergence-aware operator-wise Mixed-precision training. *CoMP* uses two-stage precision adjustment based on epochs and batches to ensure convergence and performance respectively. After that, *CoMP* performs subsequent training according to the searched optimal operator-wise mixed-precision plan. The experimental results on A100 GPU show that *CoMP* achieves a maximum performance speedup of 1.15× compared with PyTorch AMP implementation, while also saving up to 29.81% of GPU memory.

Keywords GPU · Mixed-precision · Neural network training · Auto-tuning · Performance optimization

1 Introduction

In recent years, DL has profoundly affected all walks of life and opened up new prospects for many fields, from computer vision (Li et al. 2019) and natural language processing (Lan et al. 2019), to recommendation system (Goodfellow et al. 2020). DL needs to undergo a certain amount of “learning", where trained neural networks can perform “inference" to achieve judgment and generation functions. However, neural network training has huge demands on computing power and memory (Sevilla et al. 2022). An important research issue is how to compress the model and accelerate training based on hardware characteristics, thereby facilitating promotion.

Decimals are represented by floating-point numbers in computers. Different areas have their own standards for the distribution range and precision of the values to be represented. For example, scientific computing pursues high precision, such as FP64, to perform the calculation process. Whereas in some cases (e.g., DL), such precise numerical expression is not required. This provides space for the usage of lower precision FP32 and FP16. Using lower precision means that more data can be transmitted under limited bandwidth, and data locality can be improved to reduce cache conflicts. Furthermore, the latest GPU architectures are equipped with special units, such as tensor cores, for calculations below FP32 precision.

In the DL field, the weights and other tensors involved in the calculation process of typical layers do not require a wide range of numerical expression. Therefore, low precision is widely adopted, such as low-bit quantization in the inference stage (Gholami et al. 2022), and FP32-FP16 mixed-precision representation in the training stage (Micikevicius et al. 2017). Mixed-precision can accelerate operator calculations and reduce memory footprint, which converts certain tensors to FP16 but saves weights as FP32. During training, gradient scaling can guarantee convergence to a certain extent. Mainstream DL frameworks such as Abadi et al. (2016) and

✉ Qingxiao Sun
qingxiao.sun@cup.edu.cn

Wenhao Dai
wenhao.dai@student.cup.edu.cn

Ziyi Jia
Ziyi.Jia@student.cup.edu.cn

Yuesi Bai
yuesi.bai@student.cup.edu.cn

[1] SSSLab, Department of CST, China University of Petroleum-Beijing, Beijing, China

Published online: 31 December 2024

Paszke et al. (2019) also open interfaces for mixed-precision plans for user convenience.

For mixed-precision training, DL frameworks commonly use static lists to give the precision setting for each operator. This approach takes into account expert knowledge to ensure the efficiency of mixed-precision, but it cannot always adapt to model training due to the massive possible operator combinations. For example, aggressive low-precision selection may cause numerical overflow of the operators, whereas an overly conservative high-precision selection has no room for acceleration. In addition, the computing capability of low-precision cores in GPU hardware architectures varies, further introducing format conversion costs. Recent work (He et al. 2022) has considered the conversion cost, but it requires offline training of performance models to determine the precision selection for a specified network structure. When new operators are introduced, the performance model needs to be re-trained, which cannot keep up with the development of DL frameworks.

According to our observations, the current mixed-precision methods have the following three limitations: ***1) The tradeoff between model convergence and training performance is not considered.*** Format decisions under certain operators, datasets, and tensor dimensions may lead to precision explosion. ***2) Precision decisions are not flexible enough to match various cases.*** Over-reliance on expert knowledge may affect the precision selection of operators and lead to sub-optimal decisions. ***3) The high offline cost makes it unfriendly to both hardware and frameworks.*** Large amounts of offline performance analysis are required to perform precision screening, resulting in poor performance after the updated network structure.

This paper proposes *CoMP*, an efficient operator-wise mixed-precision neural network training framework that ensures convergence. *CoMP*'s first stage determines the operators that tend to have a greater impact on convergence based on expert knowledge, and samples low-precision formats epoch by epoch to assign them to the operators with wider distribution in the network structure. Taking the loss value of a single epoch of FP32 as the basis, *CoMP* ensures that the loss value of key operators after conversion to lower precision (e.g., FP16) is within an acceptable range. Next, the fastest plan within the acceptable loss range will serve as the initial value for the second stage based on batch sampling. Since the precision decisions of key operators in the first stage ensure the model convergence, *CoMP* enumerates the precision settings of other operators according to specific rules in this stage. *CoMP* conducts profiling on a single batch at a low cost to determine the accuracy plan with the best performance.

The two-stage design of *CoMP* enables it to work with DL frameworks in a non-intrusive manner. *CoMP* comprehensively considers model convergence and training performance to determine the most appropriate precision plan. Specifically, this paper makes the following contributions:

- We propose an epoch-based sampling strategy to ensure convergence. In Stage 1, only operators that have a greater impact on convergence and meet the given loss conditions are adjusted to find a relatively good precision plan.
- We design a batch-based sampling strategy to search for better precision plans. In Stage 2, the optimal precision plan for training performance is obtained at the low cost of running only one epoch. This process has considered information such as the dataset and model structure.
- We implement the *CoMP* framework and conduct detailed performance analysis. The experimental results show that compared with PyTorch AMP, *CoMP* achieves a maximum performance speedup of 1.15×, also saves memory by 29.81% without accuracy degradation.

The rest of this paper is organized as follows. Section 2 presents the background. Section 3 presents the details of *CoMP*'s two-stage methodology. Sections 4 and 5 present the evaluation results of *CoMP* and the related work. Section 6 concludes this paper.

2 Background

2.1 Floating point precision

In the DL framework, if it is not explicitly specified, the data precision is generally single precision (FP32). In addition to FP32, there are some other floating point expressions depending on the range and accuracy of the expression, as shown in Fig. 1. In the single precision 32-bit format, 1 bit is used as a sign to indicate whether the number is positive or negative, 8 bits are reserved for the exponent to express the

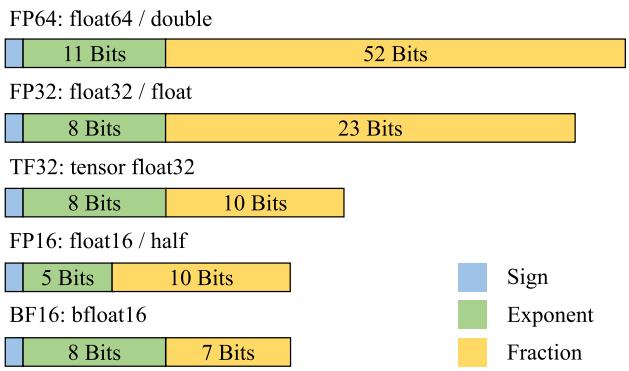

Fig. 1 Five expressions of floating point precision

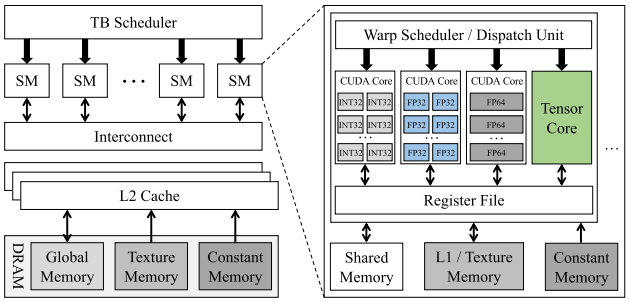

Fig. 2 The basic GPU hardware architecture with different types of processing units

numerical range, and the remaining 23 bits are used to indicate the exact precision of the floating-point number, called fraction digits. For FP64, the exponent reserves 11 bits, and the fraction digits are 52 bits, significantly expanding the range and size of the numbers that can be represented. FP16 has a narrower range, with only 5 bits for the exponent and 10 bits for the fraction digits. BF16 and TF32 have also been developed to deal with different tasks and application scenarios. BF16 uses 1/10 precision to obtain a value range of 10^{34} times without changing the memory usage of FP16. The difference between BF16 and FP32 is mainly in the fraction digits, so the conversion between BF16/FP32 is straightforward. The exponent bits of TF32 are consistent with FP32, and the fraction bits are consistent with FP16. Lower floating point precision, such as FP8, is not yet supported in stable versions of DL frameworks.

Mostly, model training uses FP32 to store weights to take advantage of a wider dynamic range. But for model inference, using lower precision integers, such as INT8/INT4, can effectively reduce latency and improve response speed. The technique of using a lower bit width than floating point precision to perform calculations and store tensors is called quantization (Gholami et al. 2022). Quantized models use reduced precision tensors instead of floating point values when performing some or all operations. Currently, at the general framework level, mainstream DL frameworks do not support quantized inference on GPUs. Still, for developers, some development packages provided by GPU manufacturers, such as TensorRT ,[1] support quantization.

2.2 GPU Hardware architecture

Figure 2 shows the GPU hardware architecture with multiple precision processing units and Tensor Core (TC) for tensor calculations. Depending on the specific model, it contains dozens to hundreds of Streaming Multiprocessors (SM). When the kernel is launched on the host side, thousands of threads will be created on the device side (GPU) according to the task planning settings. For the NVIDIA series of GPUs, every 32 threads are organized into a warp, which determines the synchronization of this group of threads. At the same time, the number of warps in the thread block (TB) will be specified in the task planning, and the TB scheduler maximizes the occupancy ratio of the GPU multi-core under the constraints of hardware resources.

The storage hierarchy on the GPU includes global memory, texture memory, constant memory, local memory, shared memory, and register file. The data in registers is the fastest accessible to the kernel; once the kernel function uses more registers than the hardware limit, data will be obtained from the cache and local memory. Compared with the data precision of FP64, lower precision (FP32 or FP16) usually means a wider addressing range in registers and caches with limited bandwidth, thereby improving the spatial locality of data. This is an important reason why mixed-precision programs achieve acceleration effects.

An SM contains multiple warp schedulers and dispatch units for scheduling threads. Starting from the Volta architecture, TC has been added to the NVIDIA series of graphics cards in addition to the traditional CUDA cores. As shown in Fig. 2, taking NVIDIA A100 as an example, its SM contains CUDA cores that support INT32, FP32, and FP64, while TC supports mixed-precision operation of FP32/FP16 and also provides support for DL inference of INT8. If the GPUs do not support TC, the DL framework (e.g., Abadi et al. (2016), Paszke et al. (2019)) use CUDA cores to perform low-precision calculations. Although mixed-precision training can still reduce memory bandwidth requirements and improve computing efficiency, these advantages are relatively small compared to GPUs with TC. The DL framework will detect whether the GPU supports TC and adjust the execution strategy, but the specific operations call to TC needs to meet certain conditions, such as tensor dimensions, the number of convolutional channels, and the dimension of the linear layer must be a multiple of 8.

2.3 Mixed-precision training

Micikevicius et al. (2017) proposed mixed-precision training. Currently, advanced DL framework such as TensorFlow and PyTorch provide corresponding APIs to allow users to enable automatic mixed-precision (AMP) on GPUs with few engineering efforts. The current primary consideration for the mixed-precision plan on the DL framework is the mixture of FP32/FP16. In recent years, the DL framework has also supported BF16, and a mixture of FP32/BF16 has been added to their mixed-precision plan. However, the hardware support for BF16 is mainly on the latest GPU architectures, such as Ampere and Hopper, while there is a lack of

[1] https://developer.nvidia.com/tensorrt.

hardware support for earlier architectures like Volta. GPUs use software-emulated BF16 without hardware support for it, affecting performance improvement and even leading to worse performance. To adapt to a wider range of GPUs, *CoMP* focuses on the mixed-precision plan of FP32/FP16.

Figure 3 shows DNN training with single-precision and mixed-precision. For single-precision training, the data precision of calculation and storage is FP32, which ensures the convergence of the model. For mixed-precision training, FP32 is used to manage weights. To overcome the gradient underflow caused by the decrease in learning rate, the loss is scaled by the coefficient *loss_scale* when calculating and then unscaled back before updating the weight. At the same time, some operators still use FP32 for calculation, which is reflected in the specific implementation of the DL framework as a black and white list mechanism.

In the DL framework, mixed-precision training on GPUs is usually implemented by combining automatic precision conversion and gradient scaling. Taking PyTorch as an example, AMP for neural network training means using `autocast()` and `Gradscaler()` at the same time. With `autocast()`, the calculation precision will be automatically selected according to its black and white list mechanism. Some operators that tend to be numerically safe (e.g., `Conv2d`, `Linear`) are in the white list and will be automatically converted to FP16 during calculation. On the contrary, some operators whose calculation results are easy to incur numerical overflow (e.g., `Layer_norm`, `Softmax`) are in the black list, and will be forced to be converted to FP32 during calculation. Some operators outside the white list and black list (e.g., `Addcmul`, `Dot`) need to select the precision in a way that ensures that there is no

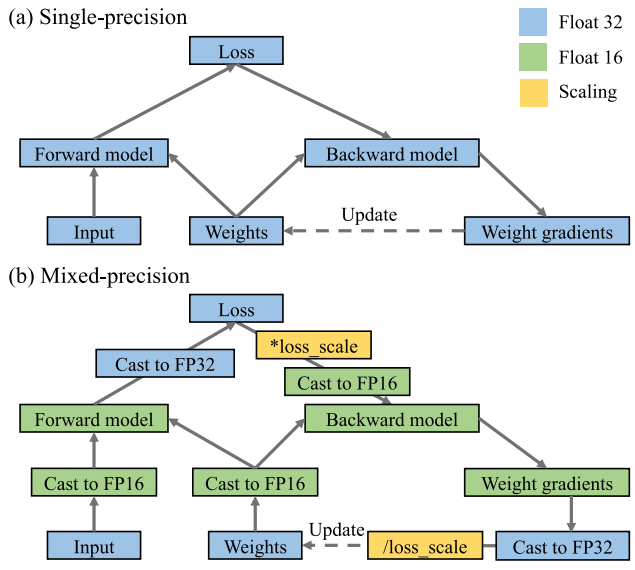

Fig. 3 Single-precision training and mixed-precision training in DL tasks

overflow in combination with the context of the operator, usually a more conservative FP32.

Similarly, in TensorFlow, users need to explicitly set a global variable of mixed-precision. Its black and white list mechanism is refined into *Allowlist*, *Denylist*, *Inferlist*, and *Clearlist*. Consequently, the current DL framework's plan for mixed-precision training predominantly focuses on execution performance and overflow prevention. Nonetheless, it overlooks nuanced adjustments for the toll of precision transformation and fails to accommodate the pronounced discrepancies in numerical distribution across models and datasets.

He et al. (2022) proposed a training framework that perceives the input tensor size and the casting cost, choosing the mixed-precision plan by considering the cost of casting FP16 and FP32 through the offline training model to predict the tensor casting cost online. This work with offline operations was intrusive in modifying the framework, which was not flexible enough in the face of the rapidly developing DL framework, and new operators were introduced. Jia et al. (2018) proposed a distributed mixed-precision plan, using FP16 for forward and backward as well as FP32 for gradients and weights. However, this method did not consider the convergence with different operators during training in detail and may not converge for some models containing special activation functions and reduction operations.

In previous methods, it was challenging to adapt to new GPU architectures and constantly updated DL frameworks at low cost. On the other hand, model training situations with different numerical distributions provide optimization space for convergence-aware mixed-precision plan adjustments. *CoMP* propose a mixed-precision training framework that adapts to the DL framework in a non-invasive way and can quickly find an efficient mixed-precision training plan that ensures convergence on modern GPUs at a relatively low cost.

3 Methodology

3.1 Design overview

CoMP is a convergence-aware operator-wise mixed-precision training framework. It mainly collects convergence and performance data through two stages of testing and then selects a plan that guarantees convergence and has the optimal performance for post-training. *CoMP*'s performance collection does not require modifying the structure of the DL framework. The optimal mixed-precision plan on the current GPU platform can be collected cheaply based on the epoch-based and batch-based sampling process.

Figure 4 shows the design overview of *CoMP*. For a model defined by a DL framework, each operator (layer)

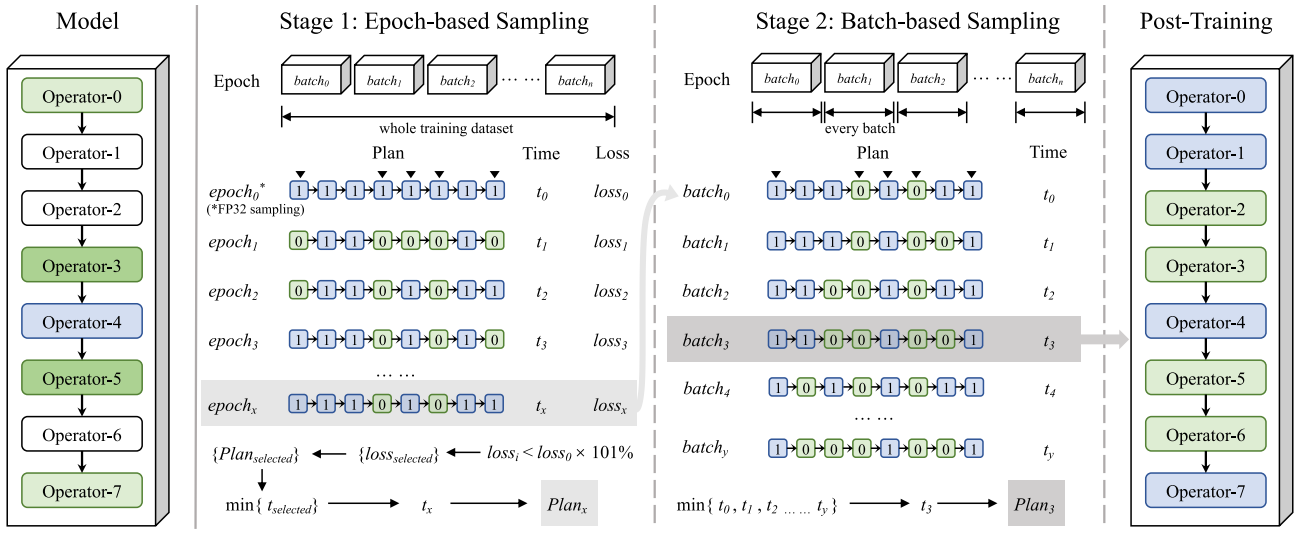

Fig. 4 Design overview of *CoMP* framework

Table 1 Operators contained in Stage1_OP_list

Operator	Precision	Description
`Conv2d` `Linear` `Matmul` `ReLU`	FP16	if the input tensor satisfies the requirement to enable TC
`Layer_norm` `Softmax`	FP32	Try converting to FP16 and perceive convergence

will tend to use FP16 or FP32 based on the list mechanism in the DL framework. *CoMP* perform sampling based on epoch in Stage 1; each mixed-precision plan must traverse the dataset thoroughly once to obtain the loss value and record the processing time. In Stage 2, convergence has been ensured, so sampling is performed based on batch. Based on the existing mixed-precision plan, other combinations of operators with little impact on convergence are enumerated to build a search space and search for the training mixed-precision plan with optimal performance.

According to the list mechanism of the DL framework and the usage of TC, *CoMP* extracts operators with clear conversion tendencies and puts them into Stage1_OP_list. As shown in Table 1, the DL framework sets the precision of operators in the whitelist (e.g., `Conv2d`) to FP16 by default. The reason is that they rarely cause value overflow, and *CoMP* will further check whether the input tensor satisfies the requirement to enable TC. For the operators in the blacklist such as `Softmax`, although the DL framework conservatively keeps them as FP32, *CoMP* will try converting to FP16 and perceive convergence.

Then, in Stage 1, based on epoch sampling, Stage 1 will first use the pure FP32 precision plan to run an epoch as a basis and then mainly selectively adjust the Stage1_OP_list to obtain different sampling policies to complete the perception of convergence. Based on the loss value of full FP32 training, on the premise that the fluctuation of the obtained loss value is acceptable, the plan with the fastest running speed is selected as the initial sampling plan of Stage 2. In the second stage, the precision of some operators determined in Stage 1 will be fixed according to the Stage1_OP_list, and other operators outside the black and white list of the DL framework will continue to be adjusted. Performance data will be quickly sampled based on batches to find the plan with the optimal performance and put it into post-training.

3.2 Stage 1: Epoch-based sampling

Stage 1 of *CoMP* is based on epoch-based convergence-aware performance sampling. In machine learning training, it usually takes dozens to hundreds of epochs to fully train the model, and each epoch means that the dataset is wholly traversed once. Running an epoch means traversing the dataset once, which can ensure the fairness of convergence perception during the sampling process and exclude the fact that the different division methods and data distribution characteristics of the dataset affect the perception of convergence. Although, in this stage, it takes an entire epoch to get feedback on a particular precision plan, it is acceptable in the overall training overhead, which is proved by the subsequent Sect. 4.5.

When using the built-in mixed-precision plan, the DL framework uses a list mechanism for the operators that make up the model. More specifically, it is determined

by whether the operator's calculation is prone to overflow or underflow; that is, it is determined by numerical safety. For example, for `Linear`, it is included in the autocast-to-FP16 list of PyTorch, which means that this operator is generally numerically safe. Still, sometimes different context precision brings additional precision conversion overhead, or the FP16 operator cannot use TC to enjoy the hardware acceleration effect due to specific dimensions, which makes the tendency to convert to FP16 unreasonable. Or for operators such as `Exp`, it exists in the autocast-to-FP32 list of PyTorch, but sometimes due to different specific parameters of the model and differences in the numerical distribution of the dataset, it does not cause the overflow expected by conservatives. These omissions in the list mechanism provide *CoMP* with room for adjustment.

For operators with clear conversion tendencies in the DL framework, Stage 1 will first obtain the information of these operators and save them in Stage1_OP_list when performing the first (basis) sampling. These operators are mainly adjusted in Stage 1, as shown in the triangle position in the Fig. 4. However, it does not mean that all combinations need to be enumerated. If the operator in the model exists in the autocast-to-FP16 list of the DL framework, then its parameters will be further checked to see if they meet the requirements of using TC. If they do, then its calculation precision will be forced to be set to FP16, reflected in the plan expressed in a string as "0", such as the dark green Operator-3 and Operator-5 on the left in the Fig. 4. If it does not meet the requirements of using TC, such as the light green part Operator-0 and Operator-7 on the left in the Fig. 4, then this may be adjusted to FP32 and needs to be tested as an optional plan in Stage 1. On the contrary, if the operator is in the autocast-to-FP32 list of the DL framework, such as Operator-4, then there is also a possible opportunity to try and generate an optimized plan.

So, for the example of Stage 1 in Fig. 4, three operators need to try different precision combinations: Operator-0, Operator-4, and Operator-7. Although Operator-3 and Operator-5 are in Stage1_OP_list, they are not tried because they can use TC. By recording the $loss_i$ and running time t_i of these operators at different precisions, and comparing them with the $loss_0$ obtained by the first sampling of Stage 1, which is wholly run with FP32, when $loss_i < loss_0 \times 101\%$, we think this is a safe range, Algorithm 1 also shows this process. We empirically determined this range through a large number of tests. The set of $loss_i$ in this range is $loss_{selected}$, and the $plan*$ with the shortest running time is selected from this setting. For the example in Fig. 4, $plan_x$ is selected.

Algorithm 1 Sampling Process for Stage 1

Input: Storing plan, loss, and time during sampling $Info_{s1}$, Stage1_OP_list OP_{s1}, Initialized FP32 plan $fp32\_plan$
Output: Plan selected by Stage 1 $result$
//Filter operators meet the conditions
$adjust\_list \leftarrow Filter\_operators(OP_{s1})$
$net\_fp32 \leftarrow model(fp32\_plan)$ // FP32 network
$loss_0, t_0 \leftarrow train(net\_fp32)$
// Enumerate binary combinations
for i in range $[1, 2^{len(adjust\_list)}]$ **do**
    $plan_i \leftarrow Generate\_plan\_string(i)$
    $net \leftarrow model(new\_plan)$ // Sampled network
    $loss_i, t_i \leftarrow train(net)$ // Train for an epoch
    $Info_{s1}.append(plan \leftarrow plan_i, loss \leftarrow loss_i, time \leftarrow t_i)$
end for
// Select $plan_i$ with $loss_i < loss_0 \times 101\%$ and the smallest t_i
$result \leftarrow Select\_plan(Info_{s1})$
return $result$

The epoch-based sampling in the first stage refers to the precision conversion tendency in the DL framework, comprehensively considers whether TC can be used, and obtains the convergence and performance of different mixed-precision plans through actual tests. Since the sampling of Stage 1 determines the accuracy of operators with poor numerical security in the DL framework, the mixed-precision plans that are prone to non-convergence are eliminated through convergence perception. At the end of Stage 1, an initial plan that determines the accuracy of operators at the corresponding positions of Stage1_OP_list will be given to Stage 2. Stage 2 will perform batch-based sampling on other operators that do not have a clear precision conversion tendency, those operators outside the exact black and white list.

3.3 Stage 2: Batch-based sampling

In the first stage of *CoMP*, operators with a certain precision conversion tendency in the DL framework are determined (i.e., the operators in Stage1_OP_list). Some of these operators are numerically safe and have the potential to use TC, while others are not numerically safe and are set to FP32 in the preliminary plan. The remaining operators do not have such a conversion tendency and need to determine the specific plan based on the context precision, such as `ReLU` and `Dropout`. These operators are interspersed and distributed in the model, Operator-1, Operator-2, and Operator-6 shown on the left side of Fig. 4. Through experimental testing, we have learned that if only a single such operator is used, the performance can be better than FP32 under FP16. However, factors such as the acceleration effect of specific

GPU hardware, the size of the input tensor, and the precision conversion cost of its context are considered. In that case, further sampling through Stage 2 is required to determine at what precision this operator can perform better for the overall model.

Since the precision of operators in Stage1_OP_list has been determined, we can consider the conversion of the context to enumerate the remaining operator precision plans. As shown in the algorithm 2, P_{s1} is the mixed-precision plan with the optimal performance within the acceptable loss range selected in the first stage. This plan is indeed determined to be the operator in Stage1_OP_list. The operator to be adjusted in Stage 2 is still the default FP32 precision. Using the operator position in Stage1_OP_list as the split point to traverse P_{s1}, when the precision of two non-adjacent operators in Stage1_OP_list is the same. For example, if both are FP32, then the operator between the two operators also maintains FP32 precision, although executing the middle operator alone may achieve better performance in FP16. When the precision of two non-adjacent operators in Stage1_OP_list is inconsistent, enumeration can be performed to adjust the precision of the operators in the middle. Such enumeration is tested to integrate the computing capability of the current GPU hardware, the cost difference of FP16/FP32 conversion, and the fixed cost of the conversion itself.

Algorithm 2 Generate Strings for Stage 2

Input: Plan string that selected by Stage 1 P_{s1}, Stage1_OP_list OP_{s1}
Output: Plan strings generated *results*
$OP_{s1}.insert(0, -1)$ // Left boundary
$OP_{s1}.append(len(P_{s1}) - 1\})$ // Right boundary
$results \leftarrow [P_{s1}]$ // Initialize with original string
for i in range $[0, len(OP_{s1}))$ **do**
  $start \leftarrow OP_{s1}[i]$
  $end \leftarrow OP_{s1}[i+1]$
  if $end > start$ and $P_{s1}[start] \neq P_{s1}[end]$ **then**
    for _ in range $[0, 2^{(end-start)})$ **do**
      // Enumerate binary combinations
      $newStr \leftarrow get\_newStr()$
      $results.append(newStr)$
    end for
  end if
end for
return *results*

Models are commonly trained by processing data in batches, iterating len(dataset) / *batch_size* times per epoch, with parameter updates at each batch. As shown in Fig. 4, there are *y* batches to iterate. The calculation results in each batch will be different due to the data distribution characteristics of the dataset, but the amount of calculation is determined by the amount of data in this batch, which is also the part that we are concerned about affecting performance. Therefore, sampling according to the batch not only ensures the fairness of different plans but also has a reasonably low testing cost. In actual operation, assigning a different precision plan to each *y* batch means defining *y* models, and each model only runs the first batch in an epoch. The first stage ensures convergence according to the sampling of the epoch, and the sampling in the second stage directly selects the plan with the best performance and maintains it into post-training.

3.4 Implementation details

CoMP is mainly based on the PyTorch framework and follows the basic requirements for building models in the DL framework. For example, the general model is defined as `class AlexNet(nn.Module)`. When building the model, the sequential programming method is used to flatten the model operators to facilitate the forced modification of the precision of each operator. In the decision-making process of *CoMP*, the specific precision setting of each operator, that is, the mixed-precision plan, is passed in the form of a "01..." string. The forward function of the model is modified to automatically insert the `x.to(torch.float16)` or `x.to(torch.float32)` function according to the plan situation to prevent data type mismatch problems during calculation. The two-stage sampling in *CoMP* is implemented in Python with 1600 LOC.

The operation and deployment of *CoMP* do not require re-modifying the source code of the DL framework, and it works with the framework in a non-intrusive manner. As shown in Fig. 5, for the AMP with built-in support in the DL framework, after loading the dataset and model, its mixed-precision plan is judged based on the black and white list mechanism, and then directly deployed on the GPU. Although there are some transparent compilation optimizations adapted to the GPU in the middle, there is no feedback tuning mechanism. The workflow of *CoMP* is to search for the optimal plan while ensuring convergence through two-stage sampling and provide an optimized operator-wise mixed-precision plan, which is put into post-training and finally deployed to the GPU for efficient mixed-precision training.

As an important reference for this work, He et al. (2022) has no open source code. Its primary function is based on TensorFlow, and it realizes graph rewriting by modifying some interfaces in the framework. It also tests some operators with different input and output sizes based on the current GPU platform in advance for performance modeling. This means that it is an intrusive way, and the DL framework is modified and needs to be recompiled instead of directly calling the dynamic library, which brings great difficulties to the migration of functions and the extension of core ideas.

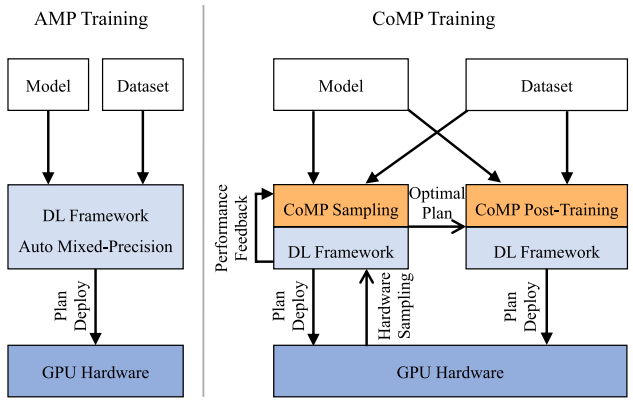

Fig. 5 Comparison of *CoMP* and AMP workflows

Therefore, we compare *CoMP* with PyTorch AMP instead, which is the mixed-precision training implementation of PyTorch. *CoMP* uses a two-stage sampling strategy to perceive the convergence and performance feedback of the current device platform. Based on following certain programming specifications in Fig. 5, the core components of *CoMP* can be deployed to other frameworks and GPU platforms.

4 Evaluation

4.1 Experiment setup

The hardware we used for the experiments is shown in Table 2. The A100, embodying NVIDIA's Ampre architecture, is extensively employed in data centers and by research entities and corporations for deploying DL training tasks. The RTX 4090, embodying NVIDIA's Ada architecture, frequently represents a more cost-effective option for developers in non-bandwidth-limited tasks. Regarding software, we use NVIDIA driver 535.98, CUDA 12.4, PyTorch 2.3.1 and DGL 2.4 Wang et al. (2019) compiled from source for the GNN model implementation. All software is installed within a docker container running Ubuntu 22.04. All comparative experiments are conducted within this docker container to ensure effective isolation.

We test a total of six models, as shown in Table 3, namely AlexNet Krizhevsky et al. (2012), Vgg16 (Simonyan and Zisserman 2014), GAN (Goodfellow et al. 2020) adopted in computer vision (CV), GCN (Kipf and Welling 2016) and GAT (Veličković et al. 2017) adopted in recommendation system (RES), and BERT-base (Lan et al. 2019) adopted in natural language processing (NLP). The datasets used for the CV models are CIFAR-100 (60,000 32×32 images) (Krizhevsky et al. 2009), MNIST (60,000 28×28 images) and ImageNet (14,197,122 224×224 images) (Russakovsky et al. 2015), the datasets used for the RES models are Cora (2,708 nodes, 5,429 edges) and Reddit (232,965 nodes, 11,606,919 edges) (Huang et al. 2021), and the dataset used for the NLP model is SQuADv2.0 (100,000 questions and over 50,000 unanswerable questions) (Rajpurkar et al. 2018). The *batch_size* of AlexNet and GAN is set to 156, whereas the *batch_size*s of Vgg16 and BERT-base are 128 and 16, respectively. We run each experiment dozens of times to reduce machine errors.

Table 2 Hardware specifications

GPU1	NVIDIA A100 PCIe 40 GB (Ampere)
CUDA Cores	6912 (108 SMs)
Tensor Cores	432
L1 Cache	192KB (per SM)
L2 Cache	40 MB
Device memory	40 GB HBM2e
GPU2	NVIDIA GeForce RTX 4090 (Ada)
CUDA Cores	16384 (128 SMs)
Tensor Cores	512
L1 Cache	128 KB (per SM)
L2 Cache	72 MB
Device memory	24 GB GDDR6X

4.2 Training performance

Tables 4 and 5 show the performance of FP32, AMP and *CoMP* applied to six models on different datasets and two GPU platforms. It can be observed that in most cases, *CoMP* can bring specific performance improvements. Compared with the standard precision FP32 implementation, *CoMP* can bring 1.28× and 1.11× speedup ratios on A100 and RTX 4090 GPUs, respectively. Compared with the most advanced framework mixed-precision implementation PyTorch AMP, *CoMP* can further bring an average speedup ratio of 1.04× and 1.02× on the two GPU platforms, respectively. *CoMP* obtains valuable performance feedback through a two-stage mechanism on the target platform, respectively using epoch-based sampling and batch-based sampling, and then selects

Table 3 Models and datasets for evaluation

Application	Model	Dataset
CV	AlexNet	CIFAR-100 ImageNet
	Vgg16	
	GAN	MNIST/ImageNet
RES	GCN	Cora Reddit
	GAT	
NLP	BERT-base	SQuADv2.0

Table 4 Training performance of CV & NLP models on A100 and RTX 4090 GPUs

Model	Dataset	Time Per Epoch (s)					
		A100			RTX 4090		
		FP32	AMP	CoMP	FP32	AMP	CoMP
AlexNet	CIFAR-100	4.54	3.84	**3.65**	2.17	2.15	**2.09**
Vgg16	CIFAR-100	8.32	5.08	**4.42**	3.89	2.53	**2.29**
AlexNet	ImageNet	1363.08	1345.41	**1340.06**	1383.36	1340.39	**1329.90**
Vgg16	ImageNet	4346.02	2895.72	**2792.61**	3671.21	2287.73	**2281.51**
GAN	ImageNet	2014.56	**1804.22**	1825.80	1398.75	**1299.09**	1303.87
GAN	MNIST	11.77	11.71	**11.51**	3.56	3.52	**3.29**
BERT-base	SQuADv2.0	1.53	0.83	**0.80**	1.37	0.93	**0.90**

The highest performance is in bold

Table 5 Training performance of GNN models on A100 and RTX 4090 GPUs

Model	Dataset	Time Per Epoch (ms)					
		A100			RTX 4090		
		FP32	AMP	CoMP	FP32	AMP	CoMP
GCN	Cora	0.68	0.71	**0.68**	0.30	0.31	**0.29**
GAT	Cora	1.03	1.12	**1.02**	0.55	**0.53**	0.55
GCN	Reddit	8.27	7.82	**7.79**	5.37	**4.38**	4.90
GAT	Reddit	186.16	155.50	**155.21**	OOM	71.11	**70.76**

The highest performance is in bold

the mixed-precision plan with optimal performance. Of course, this is based on ensuring convergence, as shown in Sect. 4.4.

Compared with the FP32 plan, the most significant acceleration effect of *CoMP* appears on the BERT-base model trained (fine-tuned) using SQuADv2.0, with a speedup ratio of 1.91× on A100 and 1.52× on RTX 4090. We believe there are more `Linear` operations in the BERT-base, and its model structure is more complex than other tested models, giving *CoMP* a more prominent search space to find a suitable mixed-precision plan. For CV models, Vgg16 training on the CIFAR-100 dataset, *CoMP* can bring a 1.88× speedup on A100 compared to the FP32 implementation, and still bring a 1.70× speedup on RTX 4090; compared with the AMP implementation, it can bring 1.15× and 1.11× speedup on A100 and RTX 4090 respectively. This further proves that *CoMP*'s search and decision-making for mixed-precision plans requires a certain complexity of the model structure to form a corresponding search space. A relatively more complex model structure often means more changes in the intermediate tensor dimensions and more types of operators that affect convergence. The conversion cost between FP32 and FP16 and whether TC can be used will affect training performance. The batch-based sampling can perceive such impacts from performance data through actual testing, thereby obtaining a relatively optimal mixed-precision plan.

When the experimental platform is determined, the same model with the same parameter settings, *CoMP* also has different acceleration effects under different datasets. For example, the experimental platform is fixed as A100 GPU, Vgg16 on CIFAR-100 and ImageNet, respectively; the acceleration effect of *CoMP* on large-scale datasets is not as strong as that on small-scale datasets. Even for PyTorch AMP, some models have no performance improvement, such as AlexNet on ImageNet and GCN on Cora. After digging deeper into PyTorch's mixed-precision plan, we find that some mixed-precision optimizations are enabled by default in FP32 implementation. However, the training performance of *CoMP* will not be worse than that of FP32 implementation, even in the worst case, returning to the situation where all operators are FP32, the performance is consistent with the benchmark. In some cases where the search space is limited, *CoMP* achieves a performance degradation of less than 10% compared to AMP.

For GPU platforms tested, RTX 4090 generally performs better than A100 because the former actually has more CUDA Cores and TCs, indicating more SM resources available. On small-scale datasets such as CIFAR-100 and MINST, RTX 4090 has a more obvious performance advantage; on larger-scale datasets such as ImageNet, this advantage is not apparent. The A100's high-bandwidth memory and larger L1 Cache, which are conducive to model inference, do not

show advantages here. It is worth noting that GAT consumes large GPU memory when running on larger datasets (such as Reddit). Specifically, the results mentioned in Sect. 4.3 show that GAT consumes 26.62 GB of memory when running on Reddit, which leads to out of memory (OOM) problems on RTX 4090 with a memory specification of 24 GB. Only on A100 with a memory specification of 40 GB can training be performed without changing hyper-parameters.

4.3 GPU memory consumption

Table 6 shows the peak GPU memory usage of six models under different datasets; the results show that the *CoMP* method can save GPU memory in most cases. Overall, compared with the standard implementation of FP32, *CoMP* saves an average of 29.54%, and compared with the implementation of PyTorch AMP, it saves an average of 14.03%. Among them, AlexNet has the most obvious memory savings on the CIFAR-100 dataset, saving 29.81% compared with AMP, and 57.30% compared with the standard implementation of FP32. On the same dataset, Vgg16 also saves 28.93% of memory compared with AMP by applying *CoMP*. The *CoMP* method is mainly effective in saving GPU memory for models primarily composed of convolution operations, such as AlexNet, Vgg16 and GAN. However, for GNN models, the space that can be saved by adjusting the mixed-precision plan is limited because of the DGL customized library and the simple model structure.

It is worth noting that for models trained on the ImageNet dataset, although the proportion of memory savings brought by the *CoMP* is not the largest, its absolute saving is the largest. For example, for Vgg16 trained on ImageNet, the *batch_size* is 128, then *CoMP* saves 937.84 MB relative to AMP, and saves up to 7771.04 MB compared to FP32. In absolute terms, it drops from 17201.88 MB to 9430.84 MB. This is a critical saving for some GPU processors with limited memory, which means that training can be performed without adjusting *batch_size*. *CoMP* uses convergence-aware sampling, combined with the use of TC, greedily uses FP16 for calculations, and achieves the effect of gradient scaling by promptly discarding mixed-precision plans that are prone to non-convergence, without the need for adaptive gradient scaling and scaling back. This makes *CoMP* save corresponding storage space compared to AMP, which includes automatic conversion and gradient scaling, thereby improving bandwidth utilization and data locality.

4.4 Accuracy comparison

Table 7 shows the accuracy or validation score of the model after training 100 epochs. FP32 is PyTorch's default precision for neural network training, while AMP combines precision auto-cast and gradient scaling. It can be observed that under the protection of gradient scaling, the accuracy of each model is slightly affected, and the accuracy of GAN and GNN models can be maintained almost the same. For GAN, our primary evaluation is the accuracy performance

Table 6 Peak memory utilization during model training

Model	Dataset 1	Memory Utilization (MB)			Dataset 2	Memory Utilization (MB)		
		FP32	AMP	CoMP		FP32	AMP	CoMP
AlexNet	CIFAR-100	797.80	485.31	**340.64**	ImageNet	1698.54	1463.80	**1084.58**
Vgg16	CIFAR-100	1227.38	813.91	**578.46**	ImageNet	17201.88	10368.68	**9430.84**
GAN	MNIST	37.77	37.77	**27.65**	ImageNet	613.26	613.26	**462.14**
GCN	Cora	54.41	56.83	**54.38**	Reddit	6709.75	6709.75	6709.75
GAT	Cora	130.04	**108.48**	**108.48**	Reddit	26620.03	16025.70	**16001.81**
BERT-base	SQuADv2.0	4997.26	3910.80	**3724.78**				

The lowest memory consumption is in bold

Table 7 The Top-1 accuracy or F1 validation score after model training

Model	Dataset 1	Top-1 Accuracy (%)			Dataset 2	Top-1 Accuracy (%)		
		FP32	AMP	CoMP		FP32	AMP	CoMP
AlexNet	CIFAR-100	61.97	61.90	**62.55**	ImageNet	**63.37**	62.07	62.78
Vgg16	CIFAR-100	70.04	**70.88**	70.12	ImageNet	**72.40**	72.37	72.36
GAN	MNIST	**49.95**	**49.95**	48.79	ImageNet	49.91	**49.95**	**49.95**
GCN	Cora	79.50	79.50	**79.60**	Reddit	**94.31**	94.30	94.27
GAT	Cora	**82.20**	81.20	81.90	Reddit	**78.34**	77.02	77.02
BERT-base	SQuADv2.0	**88.50**	88.20	88.00				

The highest accuracy is in bold

of the discriminator. Since the leading judgment is between machine-generated images and real images, the closer the accuracy is to 50.00%, the better the effect. The BERT-base NLP task is a special one; the validation score F1 is used for evaluation, and the stability of the score is also maintained under the *CoMP* method whose verification is within an acceptable range.

The premise for *CoMP* to obtain the optimal mixed-precision plan is to ensure convergence. When the corresponding *CoMP*'s Stage 1 samples according to epoch, each epoch actually traverses the dataset. Suppose the loss value under the current mixed-precision plan is more significant than 101% of the baseline value during sampling of a single epoch. In this case, *CoMP* will abandon the current plan and gradually return to a more conservative plan, even adjusting the precison of all operators to FP32. We set the threshold to 101% based on experience, which ensures that value overflow can be perceived without placing too strict restrictions on the value range. By doing so, *CoMP* can perceive the changes in accuracy and minimize the fluctuation caused by precision adjustment. *CoMP*'s second stage is totally based on the convergent plan to further sample according to the batch and take the plan with the optimal performance. In the worst case, *CoMP* will fall back to the precision plan of using FP32 completely, so the convergence and accuracy of the model are ensured.

4.5 Overhead analysis

CoMP obtains the optimal mixed-precision plan to ensure convergence on the current platform through a low-cost two-stage mechanism. Figure 6 shows the breakdown of these two stages in the overall training time on the A100 GPU, which includes the overhead of the *CoMP* method Stage 1, Stage 2 and the post-training process. For the above six models, the total overhead of the two stages of *CoMP* is 6.49% on average. For GAN, the overhead of *CoMP* accounts for 3.30% of the total. Because of GAN's simple structure, the number of layers of Generator and Discriminator is relatively small, resulting in a lack of diverse sampling schemes that may provide higher performance and mixed-precision plans. On the contrary, for the BERT-base model, whose model composition is quite complex, including 12 bidirectional encoder layers and the main operations are linear scaled dot product attention and `LayerNorm`, and the sampling schemes are relatively diverse, increasing the decision cost of *CoMP*. However, considering the widespread application of Transformer-based models represented by BERT-base in the industry, many manufacturers require pre-training and fine-tuning. The high performance mixed-precision plan brought by *CoMP* that guarantees convergence still has solid practical significance.

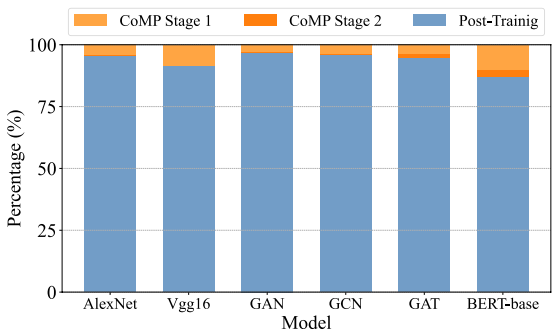

Fig. 6 Time breakdown of *CoMP* training process

For the six models we evaluated, when the hardware platform is fixed, the trend of sampling breakdown at each stage of *CoMP* operation is mainly based on the structural information of the model itself. Although the final mixed-precision plan will vary depending on the size of the internal intermediate tensor and the number of TC on the hardware platform, the sampling time of each stage is proportional to the datasets. This means that for larger datasets, the decision cost of *CoMP*'s convergence-aware sampling will also increase. But often, for larger datasets, more epochs are required to converge, and the expectation for accuracy will be higher; as a result, *CoMP* still has great potential for application on large-scale datasets.

4.6 Ablation study

Both stages involved in *CoMP* play key roles. Stage 1 eliminates overly aggressive mixed-precision plans through convergence perception to ensure the convergence of model training. Stage 2 continues to look for optimal performance plans at a low cost through batch-based sampling. We perform ablation studies on each model to compare *CoMP* with only epoch-based sampling, only batch-based sampling, and both sampling stages, showing the accuracy and performance differences.

Figure 7 shows the training performance for model training with only Stage 1, only Stage 2, and both stages. To ensure that the values in the vertical axis are distributed in similar intervals, the performance data here is run 1000 times for GCN and GAT. The acceleration effect for Vgg16 is the most obvious, whose speedup ratio of *CoMP* with two stages is 1.14× because the operator combination such as `Conv2d` and `Linear` contained in the Vgg16 model is more diverse, which provides suitable tuning space for the second stage of *CoMP*. The least obvious acceleration effect appears in BERT-base, because most of the operators in this model have conversion tendencies in the DL framework; that is, there are many operators in the Stage1_OP_list of *CoMP*, and most of them can be accelerated by TC, leaving little tuning space for *CoMP*. Because BERT-base and GAN do

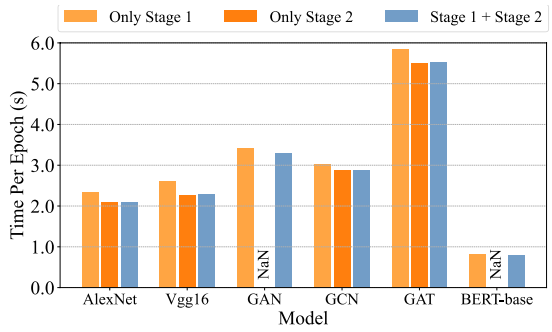

Fig. 7 The training performance of *CoMP* with only Stage 1, only Stage 2, and both stages

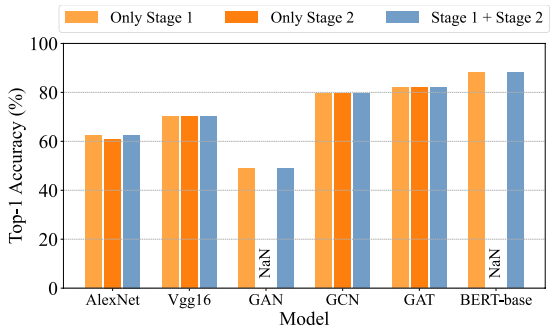

Fig. 8 Top-1 accuracy of *CoMP* with only Stage 1, only Stage 2 and both stages

not converge in *CoMP* with only Stage 2 (the result is `NaN`), the performance is meaningless and not shown in Fig. 7.

Figure 8 shows the verification accuracy of the trained model when *CoMP* has only Stage 1, only Stage 2, and both stages. For the GAN network, only Stage 2 leads to non-convergence and underflow, so there is no meaningful accuracy output. Non-convergence also occurs for BERT-large, and its F1 score is close to 0, significantly lower than that of the complete *CoMP*. For other tested models, the convergence awareness part in Stage 1 is skipped, and only batch-based performance sampling is performed. Although it achieves better training performance, it is a relatively radical plan from the perspective of the mixed-precision plan, that is, the situation where FP16 operators account for the majority.

5 Related work

5.1 Mixed-precision computation

Recently, some efforts collectively reflect the advancements in the field, where mixed-precision computation is being increasingly adopted for a wide range of applications, from ML to broader GPU-accelerated scientific computing tasks (Buttari et al. 2007; Baboulin et al. 2009; Bylina and Bylina 2013; Lam et al. 2013; Haidar et al. 2017, 2018; Gao et al. 2024; Xu et al. 2024). (Buttari et al. 2007) used a combination of FP32 and FP64 to significantly enhance the performance of many dense and sparse linear algebra algorithms while maintaining the accuracy of the results. Baboulin et al. (2009) post-processed the FP32 solution by refining it into an FP64 precision plan in the context of solving a system of linear equations. Bylina and Bylina (2013) analyzed WZ decomposition, they used a mixed-precision iterative refinement algorithm of single precision, double precision, and long double precision combined with machine learning. Kotipalli et al. (2019) designed a system to automatically select mixed-precision data types for GPU applications while ensuring adherence to specified accuracy constraints. Chitty-Venkata et al. (2022) proposed a neural structure framework that used mixed sparsity and precision search to find a mixed-precision quantization model, and searched for the global optimal sparse precision combination of each layer by using joint modeling of architecture, sparsity, and precision. Gao et al. (2024) used whether to perform automatic mixed-precision training as a tuning parameter and put into time estimator as a component of the overall scheduler. Xu et al. (2024) integrated the mixed-precision code generator and the automatic tuner by defining a parameter to achieve automatic tuning with the prediction function. Ho et al. (2021) introduced a mixed-precision algorithm framework GRAM that traded off output error and performance, with an optional half-precision math library to accelerate GPU applications such as matmul. The above work provides an important reference for *CoMP* to flexibly adjust the precision of different operators, and inspires us to use epoch-based and batch-based sampling to optimize the plan in mixed-precision neural network training.

5.2 Performance tuning on GPU

The complex computation of some applications carried out on GPUs contains certain key parameters that will significantly affect the execution efficiency of the program. For this reason, existing work has designed auto-tuning methods (Dongarra et al. 2018; Pfaffe et al. 2019; Sun et al. 2021, 2022, 2024; Randall et al. 2023; Cho et al. 2023; Parasyris et al. 2023; Zhai et al. 2023). Dongarra et al. (2018) performed batched calculation auto-tuning on GPU for a series of numerically dense linear algebra operators such as Cholesky factorization. Sun et al. (2022) designed the graph attention network (GAT) as a performance estimator, and designed a multi-head self-attention module to learn the complex relationships between features, improving the performance of DNN model compilation. Sun et al. (2021) reduced the search cost with approximate genetic algorithms

and determined the optimal parameter setting for stencil computations. On this basis, Sun et al. (2024) treated performance prediction as a regression problem and exploited pre-trained machine learning models to guide search space sampling to avoid actual execution. Parasyris et al. (2023) proposed a mechanism to facilitate the automatic scaling of large (OpenMP) offloaded applications by eliminating resource requirements and application dependencies. Zhai et al. (2023) extracted features from scheduling primitives and treated the problem as a tensor language processing task, so that the task of predicting tensor program delays through the cost model is transformed into a natural language processing (NLP) regression task. Tian et al. (2022) developed an adaptively mixed-precision (SAMP) toolkit to select the appropriate precision (FP16/INT8) to quantize the model for inference based on specific task requirements. The GPU auto-tuning mechanisms in the above work provide a reference for the mixed-precision neural network training tuning in this work. *CoMP* obtains the performance of the current platform at a low cost through a two-stage sampling and utilizes the feedback to search for the mixed-precision plan with optimal performance.

5.3 Mixed-precision architecture

Some work has designed dedicated architectures for mixed-precision to handle specific problems, such as in the field of scientific computing (Sun et al. 2008; Wang et al. 2019; Zhang et al. 2019) and machine learning (Wu et al. 2018; Nandakumar et al. 2018; Gong et al. 2019; Cai and Vasconcelos 2020; Zhou et al. 2020; Wang et al. 2021; Sun et al. 2022; Chitty-Venkata et al. 2022; Zhu et al. 2024). (Sun et al. 2008) dealt with the direct method linear solver problem on FPGAs; they proposed a mixed-precision iterative reinforcement algorithm, conducted error analysis, and implemented the designed architecture on a reconfigurable platform. Wang et al. (2019) proposed a hardware-aware automatic quantization framework that used reinforcement learning to automatically determine the quantization strategy and adopt the feedback early design loop of the hardware accelerator. Gong et al. (2019) proposed a joint optimization architecture that performs end-to-end neural network and quantization space. Through its designed search method, it found the optimal combination of architecture and accuracy (bit width), thereby directly optimizing prediction accuracy and energy consumption. Cai and Vasconcelos (2020) referred to optimal mixed-precision network search, overcame the difficulties of discrete search space and combinatorial optimization, and designed a differentiable search structure. For the important hardware acceleration unit TC of mixed-precision, Wang et al. (2021) proposed a dual-side sparse TC to exploit the sparsity of both weights and activations. Sun et al. (2022) cast a joint architectural design and quantified entropy maximization process to propose a multi-stage solution for CNNs deployed on IoT devices. Reggiani et al. (2023) proposed a collaborative design architecture of software and hardware to efficiently calculate quantized DNN convolution kernels based on byte and sub-byte data size. Zhu et al. (2024) proposed a degree-aware mixed-precision quantization scheme for GNN inference through the co-design of software and hardware, learning the appropriate bit width according to the in-degree of the node and assigning it to the node. The design of the precision plan decision mechanism of *CoMP* also has the potential to be applied to mixed-precision hardware design, requiring the coordinated support of software and hardware during compilation and runtime. This makes *CoMP* also a reference for mixed-precision architecture design and optimization.

6 Conclusion

This paper proposes *CoMP*, an efficient mixed-precision neural network training framework. Under the premise of ensuring convergence, *CoMP* tries different operator-wise precision plans through two-stage performance sampling based on epoch and batch, and finds the optimal plan for neural network training on GPU. In Stage 1, *CoMP* judges the performance of critical operators that affect convergence under different precision settings through epoch sampling and promptly avoids abnormal loss reduction solutions. In Stage 2, it ensures that the performance of other operators and key operators under the overall mixed-precision policy is enumerated at low cost through batch sampling. *CoMP*'s mixed-precision training mechanism judges the precision setting from more dimensions such as the precision conversion cost and whether to use TC, and fully combines the hardware characteristics of the current GPU platform. The experimental results show that on A100 GPU, compared with the PyTorch AMP implementation, *CoMP* can bring a maximum performance improvement of 1.15× with up to 29.81% memory saved.

In the future, we will further expand the implementation of *CoMP* to other DL frameworks such as TensorFlow and PaddlePaddle, and open source them to apply the core ideas of *CoMP* to more scenarios. Moreover, we hope to evaluate *CoMP* on other GPU platforms, such as AMD and Intel GPUs, to fully exploit the acceleration potential of *CoMP*. We also expect to continue to optimize the two-stage sampling stages of *CoMP*. For example, we can adapt more operators to the epoch-based sampling stage and implement more searching algorithms in the batch-based sampling stage. In addition, we can explore the tradeoff between model accuracy and training speed, providing a reference for subsequent DL researches.

Acknowledgements This work is supported by National Natural Science Foundation of China (Grant No. 62402525), the Fundamental Research Funds for the Central Universities (Grant No. 2462023YJRC023). Qingxiao Sun is the corresponding author.

Data availability The data that support the findings of this study are available from the first author (Wenhao Dai) upon reasonable request.

Declarations

Conflict of interest On behalf of all authors, the corresponding author states that there is no Conflict of interest.

References

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., *et al.:* Tensorflow: A system for large-scale machine learning. In: USENIX Symposium on Operating Systems Design and Implementation (2016)

Baboulin, M., Buttari, A., Dongarra, J., Kurzak, J., Langou, J., Langou, J., Luszczek, P., Tomov, S.: Accelerating scientific computations with mixed precision algorithms. Comput. Phys. Commun. **180**(12), 2526–2533 (2009)

Buttari, A., Dongarra, J., Langou, J., Langou, J., Luszczek, P., Kurzak, J.: Mixed precision iterative refinement techniques for the solution of dense linear systems. Int. J. High Perform. Comput. Appl. **21**(4), 457–466 (2007)

Bylina, B., Bylina, J.: Mixed precision iterative refinement techniques for the wz factorization. In: Federated Conference on Computer Science and Information Systems, pp. 425–431 (2013)

Cai, Z., Vasconcelos, N.: Rethinking differentiable search for mixed-precision neural networks. In: IEEE/CVF Conference on Computer Vision and Pattern Recognition, pp. 2349–2358 (2020)

Chitty-Venkata, K.T., Emani, M., Vishwanath, V., Somani, A.K.: Efficient design space exploration for sparse mixed precision neural architectures. In: International Symposium on High-Performance Parallel and Distributed Computing, pp. 265–276 (2022)

Cho, Y., Demmel, J.W., King, J., Li, X.S., Liu, Y., Luo, H.: Harnessing the crowd for autotuning high-performance computing applications. In: IEEE International Parallel and Distributed Processing Symposium, pp. 635–645 (2023)

Dongarra, J., Gates, M., Kurzak, J., Luszczek, P., Tsai, Y.M.: Autotuning numerical dense linear algebra for batched computation with gpu hardware accelerators. Proc. IEEE **106**(11), 2040–2055 (2018)

Gao, W., Zhuang, W., Li, M., Sun, P., Wen, Y., Zhang, T.: Ymir: A scheduler for foundation model fine-tuning workloads in datacenters. In: ACM International Conference on Supercomputing, pp. 259–271 (2024)

Gholami, A., Kim, S., Dong, Z., Yao, Z., Mahoney, M.W., Keutzer, K.: A survey of quantization methods for efficient neural network inference. In: Low-Power Computer Vision, pp. 291–326 (2022)

Gong, C., Jiang, Z., Wang, D., Lin, Y., Liu, Q., Pan, D.Z.: Mixed precision neural architecture search for energy efficient deep learning. In: IEEE/ACM International Conference on Computer-Aided Design, pp. 1–7 (2019). IEEE

Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., Bengio, Y.: Generative adversarial networks. Commun. ACM **63**(11), 139–144 (2020)

Haidar, A., Tomov, S., Dongarra, J., Higham, N.J.: Harnessing gpu tensor cores for fast fp16 arithmetic to speed up mixed-precision iterative refinement solvers. In: International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 603–613 (2018)

Haidar, A., Wu, P., Tomov, S., Dongarra, J.: Investigating half precision arithmetic to accelerate dense linear system solvers. In: Workshop on Latest Advances in Scalable Algorithms for Large-scale Systems, pp. 1–8 (2017)

He, X., Sun, J., Chen, H., Li, D.: Campo: Cost-aware performance optimization for mixed-precision neural network training. In: USENIX Annual Technical Conference, pp. 505–518 (2022)

Ho, N.-M., Silva, H.D., Wong, W.-F.: Gram: A framework for dynamically mixing precisions in gpu applications. ACM Trans. Architect. Code Optim. **18**(2), 1–24 (2021)

Huang, K., Zhai, J., Zheng, Z., Yi, Y., Shen, X.: Understanding and bridging the gaps in current gnn performance optimizations. In: ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 119–132 (2021)

Jia, X., Song, S., He, W., Wang, Y., Rong, H., Zhou, F., Xie, L., Guo, Z., Yang, Y., Yu, L., et al.: Highly scalable deep learning training system with mixed-precision: Training imagenet in four minutes. arXiv preprint arXiv:1807.11205 (2018)

Kipf, T.N., Welling, M.: Semi-supervised classification with graph convolutional networks. arXiv preprint arXiv:1609.02907 (2016)

Kotipalli, P.V., Singh, R., Wood, P., Laguna, I., Bagchi, S.: Ampt-ga: automatic mixed precision floating point tuning for gpu applications. In: ACM International Conference on Supercomputing, pp. 160–170 (2019)

Krizhevsky, A., Hinton, G., et al.: Learning multiple layers of features from tiny images (2009)

Krizhevsky, A., Sutskever, I., Hinton, G.E.: Imagenet classification with deep convolutional neural networks. Adv. Neural Inform. Process. Syst. (2012). https://doi.org/10.1145/3065386

Lam, M.O., Hollingsworth, J.K., Supinski, B.R., Legendre, M.P.: Automatically adapting programs for mixed-precision floating-point computation. In: ACM International Conference on Supercomputing, pp. 369–378 (2013)

Lan, Z., Chen, M., Goodman, S., Gimpel, K., Sharma, P., Soricut, R.: Albert: A lite bert for self-supervised learning of language representations. arXiv preprint arXiv:1909.11942 (2019)

Li, S., Song, W., Fang, L., Chen, Y., Ghamisi, P., Benediktsson, J.A.: Deep learning for hyperspectral image classification: an overview. IEEE Trans. Geosci. Remote Sens. **57**(9), 6690–6709 (2019)

Micikevicius, P., Narang, S., Alben, J., Diamos, G., Elsen, E., Garcia, D., Ginsburg, B., Houston, M., Kuchaiev, O., Venkatesh, G., et al.: Mixed precision training. arXiv preprint arXiv:1710.03740 (2017)

Nandakumar, S., Le Gallo, M., Boybat, I., Rajendran, B., Sebastian, A., Eleftheriou, E.: Mixed-precision architecture based on computational memory for training deep neural networks. In: IEEE International Symposium on Circuits and Systems, pp. 1–5 (2018)

Parasyris, K., Georgakoudis, G., Rangel, E., Laguna, I., Doerfert, J.: Scalable tuning of (openmp) gpu applications via kernel record and replay. In: International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 1–14 (2023)

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., et al.: Pytorch: An imperative style, high-performance deep learning library. Adv. Neural Inform. Process. Syst. **32** (2019)

Pfaffe, P., Grosser, T., Tillmann, M.: Efficient hierarchical online-autotuning: a case study on polyhedral accelerator mapping. In: ACM International Conference on Supercomputing, pp. 354–366 (2019)

Rajpurkar, P., Jia, R., Liang, P.: Know what you don't know: Unanswerable questions for squad. arXiv preprint arXiv:1806.03822 (2018)

Randall, T., Koo, J., Videau, B., Kruse, M., Wu, X., Hovland, P., Hall, M., Ge, R., Balaprakash, P.: Transfer-learning-based autotuning using gaussian copula. In: ACM International Conference on Supercomputing, pp. 37–49 (2023)

Reggiani, E., Pappalardo, A., Doblas, M., Moreto, M., Olivieri, M., Unsal, O.S., Cristal, A.: Mix-gemm: An efficient hw-sw architecture for mixed-precision quantized deep neural networks inference on edge devices. In: IEEE International Symposium on High-Performance Computer Architecture, pp. 1085–1098 (2023)

Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M., Berg, A.C., Fei-Fei, L.: Imagenet large scale visual recognition challenge. Int. J. Comput. Vision **115**(3), 211–252 (2015)

Sevilla, J., Heim, L., Ho, A., Besiroglu, T., Hobbhahn, M., Villalobos, P.: Compute trends across three eras of machine learning. In: International Joint Conference on Neural Networks, pp. 1–8 (2022)

Simonyan, K., Zisserman, A.: Very deep convolutional networks for large-scale image recognition. arXiv preprint arXiv:1409.1556 (2014)

Sun, Q., Liu, Y., Yang, H., Jiang, Z., Liu, X., Dun, M., Luan, Z., Qian, D.: cstuner: Scalable auto-tuning framework for complex stencil computation on gpus. In: IEEE International Conference on Cluster Computing, pp. 192–203 (2021)

Sun, Q., Zhang, X., Geng, H., Zhao, Y., Bai, Y., Zheng, H., Yu, B.: Gtuner: Tuning dnn computations on gpu via graph attention network. In: ACM/IEEE Design Automation Conference, pp. 1045–1050 (2022)

Sun, J., Peterson, G.D., Storaasli, O.O.: High-performance mixed-precision linear solver for fpgas. IEEE Trans. Comput. **57**(12), 1614–1623 (2008)

Sun, Z., Ge, C., Wang, J., Lin, M., Chen, H., Li, H., Sun, X.: Entropy-driven mixed-precision quantization for deep network design. Adv. Neural. Inf. Process. Syst. **35**, 21508–21520 (2022)

Sun, Q., Liu, Y., Yang, H., Jiang, Z., Luan, Z., Qian, D.: Adaptive auto-tuning framework for global exploration of stencil optimization on gpus. IEEE Trans. Parallel Distrib. Syst. **35**(1), 20–33 (2024)

Tian, R., Zhao, Z., Liu, W., Liu, H., Mao, W., Zhao, Z., Yan, K.: Samp: A toolkit for model inference with self-adaptive mixed-precision. arXiv preprint arXiv:2209.09130 (2022)

Veličković, P., Cucurull, G., Casanova, A., Romero, A., Lio, P., Bengio, Y.: Graph attention networks. arXiv preprint arXiv:1710.10903 (2017)

Wang, K., Liu, Z., Lin, Y., Lin, J., Han, S.: Haq: Hardware-aware automated quantization with mixed precision. In: IEEE/CVF Conference on Computer Vision and Pattern Recognition, pp. 8612–8620 (2019)

Wang, Y., Zhang, C., Xie, Z., Guo, C., Liu, Y., Leng, J.: Dual-side sparse tensor core. In: ACM/IEEE Annual International Symposium on Computer Architecture, pp. 1083–1095 (2021)

Wang, M., Zheng, D., Ye, Z., Gan, Q., Li, M., Song, X., Zhou, J., Ma, C., Yu, L., Gai, Y., Xiao, T., He, T., Karypis, G., Li, J., Zhang, Z.: Deep graph library: A graph-centric, highly-performant package for graph neural networks. arXiv preprint arXiv:1909.01315 (2019)

Wu, B., Wang, Y., Zhang, P., Tian, Y., Vajda, P., Keutzer, K.: Mixed precision quantization of convnets via differentiable neural architecture search. arXiv preprint arXiv:1812.00090 (2018)

Xu, J., Song, G., Zhou, B., Li, F., Hao, J., Zhao, J.: A holistic approach to automatic mixed-precision code generation and tuning for affine programs. In: ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming, pp. 55–67 (2024)

Zhai, Y., Zhang, Y., Liu, S., Chu, X., Peng, J., Ji, J., Zhang, Y.: Tlp: A deep learning-based cost model for tensor program tuning. In: ACM International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 833–845 (2023)

Zhang, H., Chen, D., Ko, S.-B.: Efficient multiple-precision floating-point fused multiply-add with mixed-precision support. IEEE Trans. Comput. **68**(7), 1035–1048 (2019)

Zhou, X., Zhang, L., Guo, C., Yin, X., Zhuo, C.: A convolutional neural network accelerator architecture with fine-granular mixed precision configurability. In: IEEE International Symposium on Circuits and Systems, pp. 1–5 (2020)

Zhu, Z., Li, F., Li, G., Liu, Z., Mo, Z., Hu, Q., Liang, X., Cheng, J.: Mega: A memory-efficient gnn accelerator exploiting degree-aware mixed-precision quantization. In: IEEE International Symposium on High-Performance Computer Architecture, pp. 124–138 (2024)