# QSync: Quantization-Minimized Synchronous Distributed Training Across Hybrid Devices

Juntao Zhao[1], Borui Wan[1], Yanghua Peng[2], Haibin Lin[2], Yibo Zhu[2], Chuan Wu[1]
[1]The University of Hong Kong, Hong Kong, [2] ByteDance Inc., USA
juntaozh, wanborui, cwu@cs.hku.hk, pengyanghua.yanghua, haibin.lin@bytedance.com, bobzhuyb@gmail.com

***Abstract*—A number of production deep learning clusters have attempted to explore inference hardware for DNN training, at the off-peak serving hours with many inference GPUs idling. Conducting DNN training with a combination of heterogeneous training and inference GPUs, known as hybrid device training, presents considerable challenges due to disparities in compute capability and significant differences in memory capacity. We propose QSync, a training system that enables efficient synchronous data-parallel DNN training over hybrid devices by strategically exploiting quantized operators. According to each device's available resource capacity, QSync selects a quantization-minimized setting for operators in the distributed DNN training graph, minimizing model accuracy degradation but keeping the training efficiency brought by quantization. We carefully design a predictor with a bi-directional mixed-precision indicator to reflect the sensitivity of DNN layers on fixed-point and floating-point low-precision operators, a replayer with a neighborhood-aware cost mapper to accurately estimate the latency of distributed hybrid mixed-precision training, and then an allocator that efficiently synchronizes workers with minimized model accuracy degradation. QSync bridges the computational graph on PyTorch to an optimized backend for quantization kernel performance and flexible support for various GPU architectures. Extensive experiments show that QSync's predictor can accurately simulate distributed mixed-precision training with $< 5\%$ error, with a consistent $0.27 - 1.03\%$ accuracy improvement over the from-scratch training tasks compared to uniform precision.**

## I. Introduction

Production AI clouds typically include both training clusters and inference serving clusters: the former consists of GPU servers equipped with training GPUs (e.g., NVIDIA A100, V100) and the latter of servers with inference GPUs (e.g., NVIDIA T4, A10). The training cluster runs throughput-sensitive deep neural network (DNN) training jobs, while the inference clusters serve latency-intensive inference tasks with strict service level objectives (SLO).

Load on an inference serving cluster often exhibits strong daily patterns, with near-full-capacity consumption at daily peak hours and low usage valleys ($< 40\%$) at an off-peak time. On the other hand, training jobs in the training cluster often experience long queuing times. To expedite training jobs and improve the utilization of inference GPUs, **hybrid-device training**, i.e., training using a mixture of training and inference GPUs has been proposed for exploiting unused resources in inference clusters to run training jobs.

Studies have addressed heterogeneous training, emphasizing either workload reallocation or elastic training methods. HetPipe [1] proposes a novel synchronized pipelined-parallel

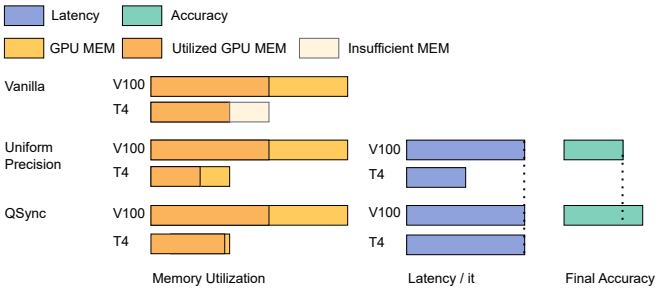



Fig. 1. Illustration of QSync. QSync reduces the number of unnecessary quantized operators without sacrificing the overall training efficiency to recover model quality.

training approach to attain optimal workload balance. In contrast, AccPar [2] concentrates on achieving equilibrium in the operator partition across devices in tensor-parallel training scenarios. Conversely, Aryl [3] implements a resource scheduling strategy that incorporates spare resources on inference GPUs to effectively execute training tasks.

However, the former approach heavily relies on the parallelism structure inherent to the training jobs and exhibits heightened sensitivity to communication bandwidth. Consequently, its suitability for data-parallel training jobs remains inadequate. In contrast, the latter approach is suitable for conventional heterogeneous device training scenarios. However, hybrid devices differ from normal case heterogeneous training with significant computation and memory discrepancies employed (refer to Sec. II). As a result, employing the same training setups (e.g. batch size) for both inference and training GPUs proves arduous when adapting to elastic methods.

Dynamic or variable batch sizing [4] is another approach to handling resource heterogeneity in data-parallel training tasks, which allocates a small local batch size to devices with smaller memory and computation capacities and a large batch size to the high-capacity ones, to balance the workload and training time. However, some operators (e.g. BatchNorm [5]) and training hyperparameter set-up rely on the local batch size, e.g., the learning rate (lr) linearly scales with the batch size [6]. Different batch size settings can significantly hurt the learned model quality (Sec. II-A).

To circumvent the challenges associated with dynamic batch sizing, a promising approach entails the utilization of quantized operators. By employing low-precision computation and storage on inference GPUs, we can effectively mitigate memory requirements and minimize the disparity between

inference and training GPUs, while simultaneously preserving the integrity of the local batch size to ensure accuracy. However, simply adopting a uniform low-precision (e.g., INT8) on inference GPUs may introduce much model accuracy degradation [7]. A good trade-off between training efficiency and model accuracy should be carefully achieved by strategically selecting the precision for each operator on inference GPUs.

We propose a *quantization-minimized synchronous* training system, QSync, that conducts effective *hybrid mixed-precision training*, i.e., different GPUs hold different precision setups to train the same full precision model, over heterogeneous devices with minimal model accuracy degradation. As illustrated in Fig. 1, instead of using uniform low-precision for all computation operators in inference GPUs, QSync intends to **convert only necessary computation operators to their low-precision counterparts**. i.e., QSync recovers the redundant low-precision operators in uniform low-precision quantization. The redundant low-precision operators refer to the operators that can be recovered to their higher bit-width representations, to improve the final model accuracy (Sec. IV) while maintaining the global training throughput without introducing new overhead.

The contributions of QSync are summarized as follows.

▷ We design a predictor that models the sensitivity of quantizable operators and accurately predicts the end-to-end latency of hybrid mixed-precision training. By applying stochastic quantization, we extend the previous theory to guarantee the convergence of the hybrid mixed-precision training, and give a proper model perturbation [8] indicator on different low-precision operators based on it. Through profiling, the predictor carefully models the casting cost (cost of converting tensors between different precisions) and tackles neighboring dependent cascading precision change for operators. Experiments show that QSync's predictor gives an indicator of precision selection that outperforms the existing schemes and can accurately simulate hybrid mixed-precision training with $< 5\%$ average error in terms of throughput prediction.

▷ We design an efficient allocator to assign precisions to different operators on heterogeneous devices, achieving quantization-minimized distributed synchronous training. The allocator searches operators' precision settings starting from the fastest available precision setup that minimizes the local model execution latency under the device memory constraints. Based on the perturbation indicator, the allocator then recovers part of the operators' precision with a higher bit. Experiments show that with the allocator, unnecessary low-precision operators on inference GPUs can be recovered, with up to 27% overall training efficiency gain compared to dynamic batch sizing and up to 1.03 % model accuracy improvement to the uniform low-precision scheme.

▷ We bridge the computation graph of QSync's from PyTorch to our own customized backend, named *LP-PyTorch*, which supports and promotes data type versatility for CUDA training. LP-PyTorch provides templated and tunable access to the underneath training kernels (e.g., CUTLASS / CUDNN [9]). It supports most of the existing GPU structures and data versatility. It further optimizes the pipelining of fixed-point kernel execution and achieves $> 10\%$ end-to-end performance gain for the INT8 training. The code is available at https://github.com/bytedance/QSync.git.

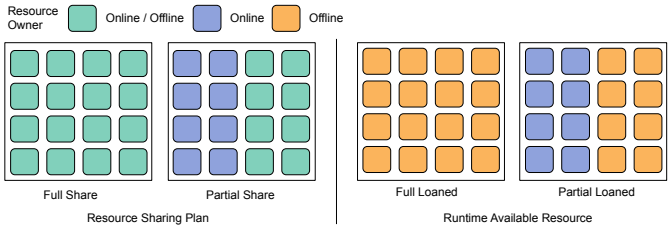



Fig. 2. Full and partial resource sharing. **Left**: Full-sharing GPU has no strict resource isolation but the partial share has a strict resource reservation. **Right**: In training, the resource on the full-sharing inference GPU can be fully utilized for the training job. As opposed to this, in partial resource sharing, only a portion of the resource is made available.

## II. Background and Motivation

### A. Hybrid-device training

In production AI systems, training clusters, which run resource-intensive DNN model training jobs, are typically heavily loaded at all times. Inference serving clusters, which serve latency-sensitive online model queries, commonly exhibit daily usage patterns according to peak/off-peak hours of applications driven by the DNN models [3].

**Hybrid training** can be considered as a distinctive instance of heterogeneous training, where both training resources and underutilized resources from inference clusters are leveraged for executing DNN training tasks. To share GPU resources in inference clusters, **full-sharing** provides the whole GPU for training jobs, while **partial-sharing** preserves some GPU resources (e.g., memory, GPU threads) for online inference serving with the rest for training jobs. An illustration is given in Fig. 2. Isolation of resources in the partial-sharing mode is typically achieved through Multi-Process Service (MPS) [10]. Typically, the low-caliber inference GPU has much lower memory and compute capability compared with the training GPU, and partial isolation makes the situation much worse.

Existing research advocates for heterogeneous training, proposing approaches such as pipelined parallelism (PP) synchronization [1] and tensor parallelism (TP) for partitioning tensors among heterogeneous devices [2]. However, it is worth noting that these approaches heavily rely on specific parallelism structures, such as parameter server and pipelining for HetPipe, and Tensor Parallel for the AccPar. Consequently, adapting these methodologies to alternative parallelism structures, such as data parallelism, necessitates substantial effort and modifications. Furthermore, while TP and PP demonstrate commendable performance for large-scale models, they tend to impose higher communication requirements and exhibit heightened sensitivity to bandwidth limitations, particularly when devices are distributed across different clusters.

Other studies, exemplified by Aryl [3], approach the training workload as individual jobs and focus on scheduling these workloads across a combination of training and inference

TABLE I
CAPABILITY OF DIFFERENT DEVICES

| GPU | FP32 TFLOPS | FP16 TFLOPS | INT8 TOPS | Memory |
|---|---|---|---|---|
| T4 | 8.1 | 65 | 130 | 16G |
| V100 | 15.7 | 125 | / | 32G |

devices. For instance, they achieve this by scaling the number of workers while maintaining a fixed local batch size across the devices. However, it is imperative to note that in the context of hybrid-device training, the training and inference GPUs (e.g., NVIDIA V100 vs. T4) exhibit significant disparities in terms of memory capabilities, as evidenced in Table I. Furthermore, this discrepancy is exacerbated by the sharing configuration depicted in Fig. 2. A training setup, such as a specific batch size, which is compatible with a training GPU, may not directly translate to an inference GPU. The mismatch can result in memory overflow or give rise to substantial synchronization bubbles, thus squandering the resources of the training GPU.

Dynamic batch sizing [4] handles a heterogeneous training environment by adjusting the batch sizes according to device capacities. While maintaining a constant global batch size, devices with higher capacities handle larger local batch sizes while low-capacity devices process data of smaller batch sizes, to achieve load-balancing. However, a key issue is not addressed in the existing dynamic batch sizing designs, changing the batch size may influence the training semantics such as convergence efficiency and final model accuracy. For example, batch normalization (BN) collects and updates statistical information within a batch, and its results depend heavily on how the data is grouped into batches; the hyperparameter settings (e.g., momentum $\lambda$) and the statistical result of the moving average in BN (running mean and running variance) highly depends on the batch size [11]. To address this problem, some works use sync-bn [12], which forces a synchronization among the statistical result above, but introduces additional synchronization overhead. As a result, expertise is required to adjust the original setting, such as a starting learning rate, a learning rate scheduler, and even incorporating an additional model structure. Our experiment in Sec.VII demonstrates that when using the learning rate adaptation setting proposed by existing works [4], dynamic batching still results in significant degradation for from-scratch training but also decreases the overall training throughput.

**Opportunity: Using mixed-precision operators for hybrid-device training without changing the batch size**.

To keep the batch size settings unchanged, an alternative way is compression. In particular, quantized distributed training (QSDP) is widely studied. Quantization compresses model weights and activations by mapping high-precision values to low-precision equivalents, which saves the memory required by the model weight and activation but also speeds up the training process. Table I shows the tera (floating point) operations per second (TOPS / TFLOPS) of operations at different precisions on NVIDIA T4 and V100 GPUs [13], [14]. The TOPS increases when the precision is halved, exhibiting substantial computation acceleration by using low-precision operators. This is also true on other chips such as NVIDIA A10 and A100. In the realm of DNN training, FP16/BF16 automated mixed-precision training has found widespread application across various tasks. Furthermore, QSDP has made remarkable progress by pushing the boundaries to include int8 quantized distributed training [15]. While existing distributed quantized training methods have successfully reduced memory requirements and expedited the training process, the straightforward uniform quantization of weights can lead to compromised theoretical convergence. Consequently, this compromises the accuracy and convergence rate [7].

Existing quantized distributed training methods uniformly quantize all main operators (linear, conv) to the same precision across different devices, which is insufficient when facing *hybrid-device training*. Firstly, we typically share a training job with a batch size that conforms to the training GPU. This ensures that the training GPU always has enough memory to hold the batch and eliminates the need to quantize the operators on it, also, some lower-precision (e.g. INT8) may be not supported by the training hardware (e.g. V100), as shown in Table I. Secondly, we only need to perform the necessary quantization on the inference GPUs to keep the training efficiency while minimizing accuracy degradation.

Our objective is to leverage quantized operators to achieve memory reduction and accelerate the training process while placing a strong emphasis on preserving accuracy. Rather than applying uniform quantization to all operators, our approach focuses on selectively quantizing essential operators specifically on inference GPUs. We quantize enough operators to fit the training workload into the inference GPU but leave some operators unchanged or with higher precision to mitigate the speed differential between inference and training GPUs after quantization and improve accuracy. We term this approach as *quantization-minimized synchronous*. Our particular focus lies in data-parallel training jobs configured on training GPUs, particularly the batch size, and strive to execute them within a hybrid device environment. This endeavor presents novel challenges and opens up new opportunities for design exploration.

### *B. Challenges in quantization-minimized synchronous in hybrid-device training*

**Measurement of quantization impact on model accuracy and training efficiency.** To simultaneously consider model accuracy and training throughput induced by low-precision operators in hybrid-device training, we need to know how changes in operator precision impact model accuracy and training throughput. The accuracy impact of both floating-point and fixed-point low-precision operators must be taken into account. Previous studies [8], [16] only address one of them and focus on the forward pass in DNN training, while operator precision in backward propagation should also be considered. For throughput estimation, accurate modeling and prediction of the timeline view of the mixed-precision global training are needed. Campo [17] used performance modeling

to predict the casting cost and operational performance with low precision and introduced a cost-aware graph rewriting strategy to optimize mixed precision training. It only considers casting costs between FP32 and FP16, ignores the precision dependency between operators[1] and cannot reflect the overall model runtime in the distributed setting.

**Efficient precision allocation.** Given possible precisions and a large number of precision-adjustable operators in a DNN model, it is time-consuming to brute-forcibly search for the best setting over all feasible mixed-precision settings. For example, given INT8, FP16, and FP32 as three optional precisions and considering setting precisions for 73 linear operators in BERT or 52 Conv2D operators in a ResNet50, the search space is $3^{52}$ or $3^{73}$, respectively. Efficiently and correctly derive the optimized mixed-precision settings is challenging.

**Low-precision versatility supports**. The realization of the aforementioned advantages of quantized training relies on the effectiveness of low-precision operator kernels. The existing training framework (e.g. PyTorch, Tensorflow) usually supports FP16/FP32 CUDA training by default. To bridge operators to their extended low-precision implementations, an efficient pipeline is required to access low-precision kernels and tune their performance for different hardware. For inference serving scenarios, *kernel tempting* is discussed to tune operators for different target devices [18]. On the contrary, none of the existing frameworks support templating low-precision training kernels (e.g. backward ops), let alone adapting them to hardware for optimized performance.

## III. Overview

We propose QSync, a quantization-minimized synchronous distributed training system to enable efficient synchronous data-parallel DNN training over hybrid devices. We consider a training cluster equipped with training GPUs of the same type and an inference serving cluster with inference GPUs of the same type. A distributed DNN training job can leverage multiple training GPUs and available resources (e.g., memory, compute capability) on some inference GPUs. Fig. 2 gives an overview of the workflow of QSync. The main idea of QSync is to use low-precision formats for necessary operators on inference GPUs. The workflow of QSync goes as follows. 1) Substituting operators in a model with mixed-precision implementation for the target hardware. 2) The cost and memory requirements for the operators under different precision are collected through profiling. Statistical data, like model depth, tensor dimension, and norms are also collected by running a few iteration steps on the GPUs using smaller batch sizes. Local DFGs with communication dependency are also traced by constructing homogenous GPU sub-sets. 3)

[1]There exists CUDA Ops that promote the widest input type. For example, the precision of the *add* operator depends on the largest precisions of its inputs. If the precisions of two inputs are not the same (such as FP16 and FP32), a cast operator is added to convert the lower-precision input to the higher precision for addition. Other operators like *ReLU* that are not directly handled by the auto casting also depend on its input precision as well.

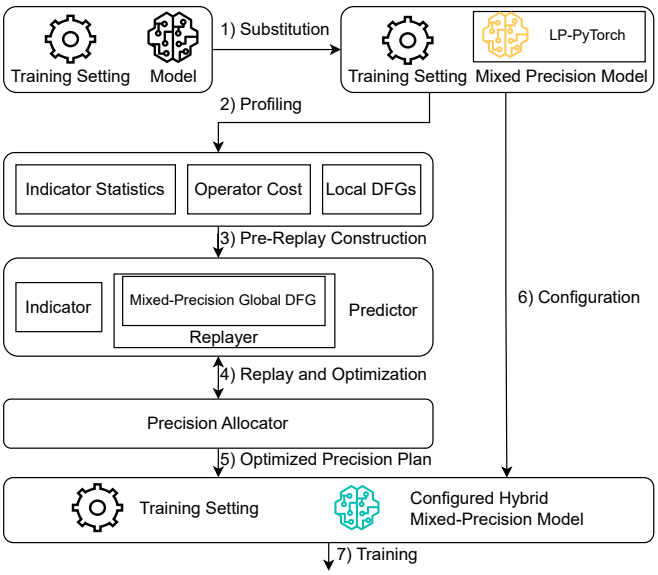



Fig. 3. QSync Workflow

The predictor calculates the indicator result (model perturbation) based on statistical data of the operators and builds a global mixed-precision data flow graph. 4) The precision allocator, with the help of the Replayer in Predictor, simulates distributed model training and estimates overall throughput under different mixed-precision settings using the guidance of the Indicator, and greedily optimizes the precision allocation plan starting from the initial global DFG. 5) The optimized precision plan is then fed back to the mixed-precision training system. 6) The mixed-precision backend then configures the low-precision kernel by selecting the best device-optimized configuration 7) Hybrid mixed-precision distributed model training is carried out using the optimized precisions.

QSync includes three main modules:

**The Predictor** is composed of an Indicator that generates operator perturbation towards precision, and a Replayer simulates local and distributed training and estimates the memory consumption and per-iteration training time.

**Precision Allocator** interacts with the predictor to search for better precision settings for operators on the inference GPUs.

**LP-PyTorch** is a backend that enhances the capabilities of deep learning frameworks like PyTorch to efficiently tune and run low-precision kernels.

## IV. The Predictor

The main problem addressed by QSync is to find an optimized precision allocation plan to operators on inference GPUs that minimizes the model accuracy degradation introduced by low-precision kernel execution while maintaining the training throughput. This formulates the problem below:

$$\min_{\{b_{io}\,|\,i\in\mathcal{K}_{inf},o\in O\}} \quad \sum_{i\in\mathcal{K}_{inf}}\sum_{o\in O}\Omega_o^{(b_{io})}$$
$$\text{s.t.} \quad \mathcal{M}_i(\{b_{io} \mid o\in O\}) \le M_i^{max}, \forall i\in\mathcal{K}_{inf}$$
$$\mathcal{E}(\{b_{ko} \mid k\in\mathcal{K}, o\in O\}) \ge T^{min} \tag{1}$$

In our hybrid-device training among $\mathcal{K}$ GPUs, $\mathcal{K}_{inf}$ is the set of inference GPUs. $\Omega_o^{(b_{io})}$ is operator $o$'s sensitivity with bit precision $b_{io}$ on inference GPU $i$, and $O$ is the set of all operators in the model directed acyclic graph (DAG). $M_i^{max}$ is the available memory capacity on inference GPU $i$. $T^{min}$ is the training throughput of the DNN training job that can be obtained using the same low-precision for all operators on inference GPUs under the memory constraints, e.g., converting all operators to int8 or fp16 depending on the lowest precision that the inference GPUs support. $\mathcal{M}_i(\cdot)$ is the predictor function that estimates the memory consumption by the training job on inference GPU $i$. $\mathcal{E}(\cdot)$ estimates overall training throughput based on the precision plan $\{b_{ko} \mid k\in\mathcal{K}, o\in O\}$ among all GPUs. Especially, $b_{ko} = 32$ on each training GPU k $\in \mathcal{K}\setminus\mathcal{K}_{inf}$. Solving problem (1) poses new challenges: 1) How to build an effective sensitivity indicator $\Omega_o^{(b_{io})}$ to measure the relationship between model accuracy degradation and operator precisions. Low-precision operators affect both forward and backward passes and can be fixed-point or floating-point, which complicates the theoretical analysis. 2) How to construct accurate predictors $\mathcal{M}_i(\cdot)$ and $\mathcal{E}(\cdot)$. Due to the casting cost between different precision and precision-dependent operators whose execution latency and memory cost depend on their inputs precision, together with the presence of a communication operator and its dependency, the end-to-end model training latency cannot be readily expressed as an independent summation of sequential operator execution costs.

When the precision of the forward operation is changed, the execution of the corresponding backward operation is also changed due to the casting. This means that precision change can lead to modifications in both the forward and backward passes of a given operator. For this reason, in this paper, we refer to an **operator** as a pair of forward and backward operations, and QSync alters the precision of forward and backward operations together.

### A. Indicator

The perturbation indicator $\Omega_o^{(b_{io})}$ qualifies the relationship between model accuracy and operator precisions. We explicitly examine the variance of the gradient of model weights introduced by the low-precision casting of operators. Large gradient variance can lead to large and unstable weight updates, making training difficult to converge. Following ACTNN [19], we analyze the convergence of our hybrid-device training when using an Unbiased Quantizer (UQ). UQ quantizes an original number to its unbiased estimation, i.e., $\mathbb{E}[UQ(x)] = x$. Stochastic rounding (SR) performs non-deterministic rounding according to the residual to nearby integer values, which is unbiased. We apply SR $\lceil\cdot\rfloor$ as our rounding method for quantization. $f(\cdot)$ denotes the loss function in model training with learning rate $\eta$. Consider the empirical-risk minimization of the loss function with parameter $\mathbf{x}\in R^d$ on a training dataset $D$. The DNN training problem can be modeled as:

$$\min_{\mathbf{x}\in R^d} f(\mathbf{x}) \coloneqq \mathop{\mathbb{E}}_{s\sim D}[f_s(\mathbf{x};\{b_{io} \mid o\in O\})], \forall i\in\mathcal{K} \tag{2}$$

where $s$ is a random sample from dataset $D$ and $f_s(\mathbf{x};\cdot)$ is the local loss function with a precision set-up for weight $\mathbf{x}$ on the GPU that processes the sample. $f_s^{(0)}(\mathbf{x})$ denotes a local loss function without low-precision operators. In QSync, we consider representative loss functions (combination) such as mean square error (MSE) and cross-entropy (CE) with softmax. Denote the input to these loss functions to be $v^{(L)}$, the ground truth to be $y^{(L)}$, the corresponding gradient function of input $v^{(L)}$ respect to loss function can be expressed as: $\nabla v^{(L)} = \gamma(v^{(L)} - y^{(L)})$ where $\gamma\in\{\frac{2}{N},\frac{1}{N},-1\}$. In QSync, the precision of these loss functions in the DNN model graph is unchanged. The following proposition states that we can obtain unbiased gradient estimation under the specification of QSync:

**Proposition 1** (Unbiased Gradient)**.** *With the loss function unchanged, by using an unbiased quantizer for linear operators , we have* $\mathbb{E}[\nabla f_s(\mathbf{x};\{b_{io} \mid o\in O\}))] = \mathbb{E}[\nabla f_s^{(0)}(\mathbf{x})]$.

Assuming SGD training, convergence proof of QSync's mixed-precision training can follow ACTNN [19]. Consider an initial model parameter $\mathbf{x}_0$ with several convergence assumptions that are widely used [20], [21].

**Assumption 1.** $\forall \mathbf{x}_t, \mathbf{x'}_t \in R^d$ *in the t-th training iteration:*
***A.1*** *($\mathcal{L}_2 - Lipschitz$)* $||\nabla f(\mathbf{x}_t) - \nabla f(\mathbf{x'}_t)|| \le L||\mathbf{x}_t - \mathbf{x'}_t||$*;*
***A.2*** *(*existence of global minimum*)* $\exists\, f^*$ *s.t.* $f(\mathbf{x}_t) \ge f^*$*;*
***A.3*** *(*bounded variance*) There exists* $\sigma^2 > 0$*, s.t.*$Var[\nabla\mathbf{x}] \le \sigma^2\forall\mathbf{x}$

**Theorem 1.** *[Convergence] Let $T$ be the maximum number of iterations. Under Assumption 1, we have*
$$\min_{t=0,1,\ldots,T-1}\mathbb{E}[\|\nabla f(\mathbf{x}_t)\|^2] \le \frac{f(\mathbf{x}_0) - \mathbb{E}[f^*]}{\sum_{t=0}^{T-1}(-\eta+\eta^2\frac{L}{2})} + \frac{\sum_{t=0}^{T-1}\eta^2\frac{L}{2}\sigma^2}{\sum_{t=0}^{T-1}(-\eta+\eta^2\frac{L}{2})}$$

Except for $\sigma$, theorem 1 has the same form as FP32 that ensures the convergence and training ability of QSync. $\sigma$ shapes the converged solution.

For a scalar $x$, with fixed-point quantization, $\bar{x} = \frac{x - z_x}{q_x}$ and $\hat{x} = \lceil\bar{x}\rfloor \times q_x + z_x$, where $\bar{x}$ is a contiguous number obtained by scaling $x$ with zero-point $z_x$ and scaling factor $q_x$, $\lceil\bar{x}\rfloor$ is the quantized scalar and $\hat{x}$ is the contiguous dequantized scalar. For floating-point quantization, [22], the value of a scalar is represented by $x = s\cdot 2^e\cdot(1+m)$, where $s, e, m$ are a sign, effective exponential bit and the mantissa. The exponents' bits are truncated and stochastic rounding is applied to the mantissa. For any vector $\mathbf{x}$, it has the following characteristics:

**Proposition 2** (Tensor Quantization Variance)**.** $Var[\hat{\mathbf{x}}] = \frac{q_{\mathbf{x}}^2 D_{\mathbf{x}}}{6}$ *for fixed-point quantization.* $Var[\hat{\mathbf{x}}] = \frac{2^{2e}\epsilon^2 D_{\mathbf{x}}}{6}$ *for floating-point quantization. $D_{\mathbf{x}}$ is the dimensionality of tensor* $\mathbf{x}$.

$\epsilon$ here is $2^{-k}$. For the IEEE standard floating-point formats, $k = 9$ for float16. We next model the variance increment of different operators in a DNN model, considering parameter, activation, and gradient in the forward and backward passes.

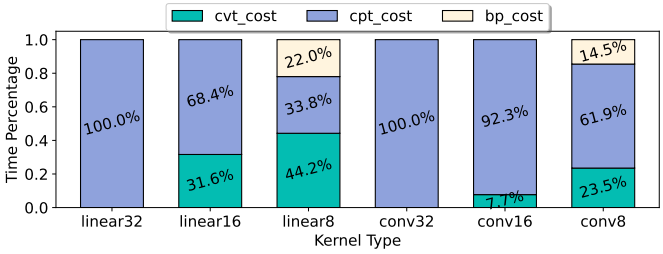



Fig. 4. Cost Composition of an Operator

**Proposition 3** (Variance Increment)**.** *We have the variance increment of operator $o$ with bit precision $b_o$, $\Omega_o^{(b_o)}$, as*

$$\Omega_o^{(b_o)} = \gamma^2 d_o \hat{\sigma}_{fp}^{(o)} + (d_L - d_o)\hat{\sigma}_{bp}^{(o)} \tag{3}$$

*Especially, for unary-input computation-intensive operators (e.g., Linear, Convolution) , we have*

$$\hat{\sigma}_{fp} = \begin{cases} \frac{1}{6}(\|\mathbf{x}\|^2 q_{\hat{\mathbf{v}}}^2 D_{\mathbf{v}} + \|\hat{\mathbf{v}}\|^2 q_{\mathbf{x}}^2 D_{\mathbf{x}}), & \textit{fixed-point quantization}, \\ \frac{1}{6}\epsilon^2(\|\mathbf{x}\|^2 2^{2e_{\mathbf{v}}} D_{\mathbf{v}} + \|\hat{\mathbf{v}}\|^2 2^{2e_{\mathbf{x}}} D_{\mathbf{x}}), & \textit{floating-point quantization} \end{cases} \tag{4}$$

$$\hat{\sigma}_{bp} = \begin{cases} \frac{1}{6}(\|\nabla\mathbf{v}\|^2 q_{\hat{\mathbf{v}}}^2 D_{\mathbf{v}} + \|\hat{\mathbf{v}}\|^2 2^{2e_{\nabla\mathbf{v}}} \epsilon^2 D_{\nabla\mathbf{v}}), & \textit{fixed-point quantization}, \\ \frac{1}{6}\epsilon^2(\|\nabla\hat{\mathbf{v}}\|^2 2^{2e_{\mathbf{v}}} D_{\mathbf{v}} + \|\hat{\mathbf{v}}\|^2 2^{2e_{\nabla\mathbf{v}}} D_{\nabla\mathbf{v}}), & \textit{floating-point quantization}. \end{cases} \tag{5}$$

Here $\mathbf{v}$ and $\nabla\mathbf{v}$ are the activation and gradient of activation for the operator $o$. From the observation of equation 4 and 5, an operator's sensitivity to the precision is determined by its depth $d_o$ with respect to the model depth $d_L$. The depth of an operator inside a model forward DAG is a measure of its distance from the root node, which can be directly obtained by applying depth-first search. Tensor dimensionality $D_{\mathbf{v}}, D_{\nabla\mathbf{v}}$ and $D_{\mathbf{x}}$, and norms $\|\nabla\mathbf{v}\|^2, \|\nabla\hat{\mathbf{v}}\|^2, \|\hat{\mathbf{v}}\|^2$ also matters. For fixed-point operators, the scaling factors of the quantization affected input and weight, $q_{\hat{\mathbf{v}}}, q_{\mathbf{x}}$, also contribute to the variance. $e_{\mathbf{v}}$, $e_{\mathbf{x}}$, $e_{\nabla\mathbf{v}}$ are effective bits, which can be derived with the data's magnitude (maximum and minimum). These factors can be collected through profiling. The norms and scaling factors $q_{\hat{\mathbf{v}}}$ and $q_{\mathbf{x}}$ are changing during training, so $\Omega_o^{(b_o)}$ of the operator is changing. Our experiments (Sec. VII-E) show that most of the relative values of factors related to the training process do not change significantly. To improve efficiency, we use the running mean of the first 50 iterations as the perturbation result of model operators, we also half the training batch size for profiling these results. Unary operators with one argument such as MaxPool does not hold learnable parameters. Their variance is only introduced in the forward pass and bounded by their input's tensor quantization variance bound. Their $\hat{\sigma}_{bp}^{(o)}$ is zero. Specifically, QSync does not modify pure matmul operations, which involve binary inputs.

### B. Replayer

**Cost Mapper.** To obtain the training throughput estimation $\mathcal{E}(\cdot)$ under different mixed-precision settings, we first analyze the cost composition of an operator at different precisions in each training iteration. Take the second last convolution in VGG16 and a regular linear operator from one of the attention blocks in BERT as examples: we execute the operator 100 times on T4 and obtain the average time composition as shown in Fig. 4 for $INT8, FP16$ *and* $FP32$. $cvt\_cost$ denotes the casting overhead in the forward pass in converting input and weight tensors to low precision, $bp\_cost$ is the additional casting overhead[2] in backward computation, and $cpt\_cost$ is the operator's execution time including both forward and backward computation. The casting cost is non-negligible with low-precision operators for all cases. A DNN model may also include operators whose precision is defined by the precisions of their inputs, e.g. add, maxpool. In some circumstances, a change in operator's precision causes a cascading precision shift among the subsequent operators, significantly changing the overall training time.

We model the costs of converting between different precisions and between floating-point and fixed-point numbers. Casting between floating-point numbers can be modeled as a linear function of tensor size [17], we focus more on shaping the fixed-point case. Fixed-point quantization requires a maximize and minimize to calculate the quantization-related scaling factor. The process has two steps: partitioning the task into thread blocks, with each thread in a block finding the maximum and minimum of a portion of the tensor data and storing the results in a shared cache. Then, a tree-like parallel reduction is applied among all thread blocks, reducing both the number of GPU threads and data simultaneously. Thus, the cost of data collection for each step can be modeled as a linear function of the tensor size. Further, runtime fixed-point quantization includes calculation of the output scaling factor; if there is no fusion for the dequantization operation on the fixed-point execution operator, a dequantization cost should also be modeled and added. Besides, different fixed-point quantization methods (e.g., channel-wise [23], layer-wise) vary in performance and result in different combinations of the dequantization methods. For example, a layer-wise quantized input and a channel-wise quantized weight should be dequantized with a channel-wise dequantizer, while a layer-wise quantized input and a layer-wise quantized weight should be dequantized with a layer-wise dequantizer. Fortunately, regardless of the dequantization type, it is essentially a kernel-level element-wise operation, so it can still be shaped as the linear cost with respect to the tensor size. In QSync, we comprehensively analyze all these scenarios and employ a collection of linear models to accurately predict the casting costs across various cases, leveraging the tensor size as a parameter.

We categorize all operators in a DNN model into two types: 1) Precision Adjustable Operators $O_{adj}$, including common computation-intensive operators, e.g. Matmul and Conv, and operators that may numerically overflow in calculation with the low-precision number, e.g. softmax. 2) Precision Dependent Operators $O_{dep}$, whose precision is determined by the

[2]Integer backward computation is shown to incur low efficiency [9]. In QSync, we perform the backward computation of fixed-point kernels in FP16, which incurs additional casting costs.

**Algorithm 1** CostMapping

1: **Input**: Local precision DAG $\mathcal{G}_i$, target operator $o$, new precision $b_{io}$, profiled op cost $CC_i$, casting cost calculator $CP$, local data flow graph $DFG$
2: **Output**: new precision DAG $\mathcal{G}_i'$ and local DFG $DFG'$
3: $\mathcal{G}_i' = UpdateDAG(\mathcal{G}_i, b_{io})$ {Update $o$'s precision}
4: $preds, succs = \mathcal{G}_i'.pred\_succ(o)$ {Get neighbors of $o$}
5: $C_i^{fwd} = 0, C_i^{bwd} = 0$
6: **for** $p \in preds$ **do**
7: **if** $b_{ip} != b_{io}$ **then**
8: $C_i^{fwd} \mathrel{+}= CP.predict(b_{ip}, b_{io}, shape_p^{output})$
9: **end if**
10: **end for**
11: **if** $o \in O_{adj}$ **then**
12: $C_i^w = CP.predict(32, b_{io}, shape_o^{weight})$
13: **else**
14: $C_i^w = 0$
15: **end if**
16: $b_{io}^{out} = output(b_{io})$ {Get operator output's precision}
17: **for** $s \in succs$ **do**
18: **if** $same(\mathcal{G}_i'.pred(s).bit) \& s \in O_{rel}$ **then**
19: $CostMapping(\mathcal{G}_i, s, b_{io}^{out}, CC_i, CP, DFG)$ {Traverse}
20: **else**
21: $UpdateFwd(s)$ {Lines 6-10: update forward casting cost}
22: **end if**
23: $C_i^{bwd} \mathrel{+}= CP.predict(b_{is}, b_{io}, shape_o^{output})$
24: **end for**
25: $C_i^{op} = CC_i[b_{io}]$ {Lookup pure operator execution cost}
26: $DFG' = UpdateDFG(DFG, C_i^{fwd}, C_i^{bwd}, C_i^w, C_i^{op})$

precision of the input provided by other operators, e.g., add and ReLU.

In QSync, we maintain three graphs to track the precision and execution timeline of training among different devices: Precision DAG, local DFG, and global DFG. For each GPU, QSync maintains a precision DAG that keeps the training model with operators' precision and its dependencies. Each GPU also has a local DFG, which is the execution line for the operator in training, further including the backward operation and optimizers. The global DFG is composed of all the local DFGs, with communication among them.

Our cost mapper updates the precision change of an operator for a certain device to its precision DAG, computes the casting cost, and fetches new pure operator execution cost for graph update, thus producing new local DFG and global DFG. The new global DFG is used for training simulation. Alg. 1 and Fig. 5 shows the procedure of cost mapping. The cost mapper first updates the target operator $o$'s precision in the precision DAG $\mathcal{G}_i$ on device $i$ (line 3), and then records the operator's predecessors, and successors and initializes the forward cost and backward cost (lines 4-5). The casting cost is estimated by the casting cost calculator. The overall casting cost of the operator with new precision in the forward pass is calculated by summing all casting costs for input has different precisions $C_i^{fwd}$ (lines 6-10). The weight casting cost $C_i^w$ is estimated based on the operator's weight shape $shape_o^{weight}$ if $o \in O_{adj}$ (lines 11-13). Next, the cost mapper traverses the successor nodes of the current operator. Suppose a successor operator is a precision-dependent operator $s \in O_{rel}$ and all the input bits of the successor the operator is the same. In that case, breath-first-search is applied to the successors to make an iterative precision change in the precision DAG with the precision of the operator output $b_{out}$ (lines 16) [3] .The predicted backward casting cost $C_i^{bwd}$ is also computed (lines 17-24). Finally, the pure operator cost is fetched from the profile result (lines 25), together with new casting costs are updated to the local $DFG'$.

**Simulator.** After updating all devices with the final precision, the simulator in the Replayer simulates the execution of the global DFG. This process includes updating the communication operator cost for all devices based on communication dependencies and using topological sort to predict the execution time of all local DFGs. To trace the communication node, we first construct distributed training on smaller homogeneous GPU sets to measure the dependencies and communication buffer size. For example, to measure communication cost on a hybrid 16 T4 + 16 V100 training, we perform 2-T4 and 2-V100 training separately and use their trace data for execution timelines. Subsequently, the communication cost was recalculated by considering the topology of hybrid training and aligning the start point of the first communication, denoted as $comm_0^{start}$. The precise estimation of the communication cost for each local DFG on device $i$ was obtained using equation (6), where $n$ represents the n-th communication operation:

$$\begin{aligned} comm_n^{start} &= max(max(\{comm_{i,n}^{start}, \forall i \in \mathcal{K}\}), comm_{i,n-1}^{end}), \forall n \in [1, N] \\ comm_{i,n}^{end} &= comm_n^{start} + max(\{comm_{i,n}^{dur}\}), \forall n \in N, \forall i \in \mathcal{K} \end{aligned} \tag{6}$$

Here $comm_{i,n}^{start/end}$, $comm_{i,n}^{dur}$ represents the duration between start/end points and 0 and duration of the n-th communication slot on device $i$. $comm_n^{start}$ is the synchronized communication start point across devices. The maximum end-to-end latency among local DFGs is then taken as the final distributed training throughput.

Training throughput estimation, $\mathcal{E}(\cdot)$, is obtained by updating the precision cost in local DFG and precision DAG, then simulating with global DFG. Additionally, memory consumption of training on device $i$, $\mathcal{M}_i(\cdot)$, can be obtained simultaneously by profiling and accumulating memory consumption based on operator precision in local precision DAG $\mathcal{G}_i$.

## V. Allocator

Based on the Indicator and the Replayer provided by the Predictor, the Allocator of QSync solves the operator precision allocation problem (1) to obtain operator precisions to use on the inference GPUs. The Allocator uses a maximum heap for each inference GPU to store indicator value differences upon precision changes. Each time, it selects the operator with the largest indicator decrement on each GPU to increase precision. Then, it estimates new overall training throughput and memory consumption using Replayer and keeps new precision if it meets memory constraints and does not decrease overall throughput.

[3]The output precision does not have to match the kernel execution precision. For example, an FP16 kernel can have an output precision of FP32 or FP16. In QSync, the output of INT8 is set to a floating point FP32.

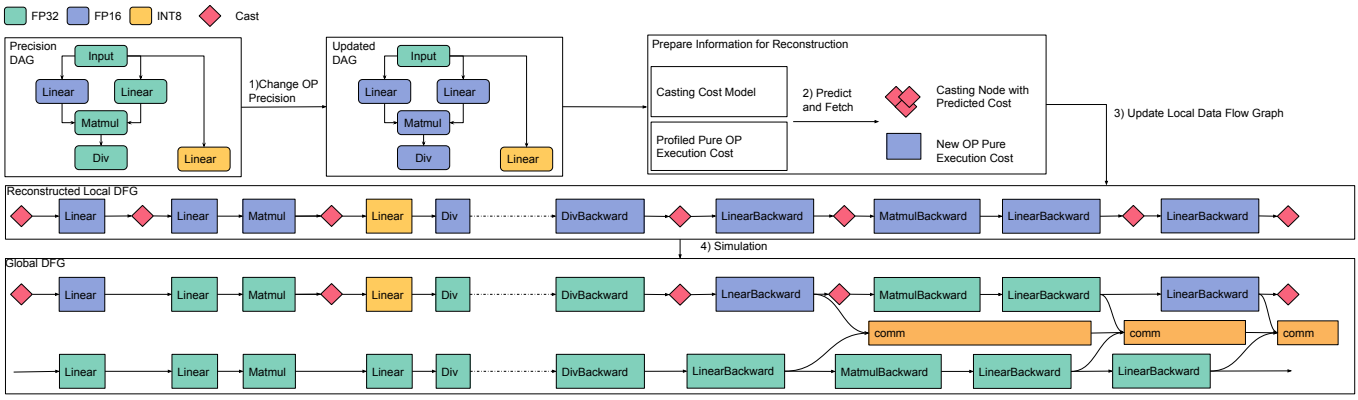



Fig. 5. Workflow of Replayer. (1) The local precision DAG is updated upon a change in operator precision, and the cost mapper traverses the graph to update the precisions of dependent operators. (2) The casting costs in the new precision DAG are calculated, and the pure operator execution cost is retrieved from the profiling results. (3) the local data flow graph (DFG) is updated, and (4) the global DFG is updated accordingly, which can be used in the overall training throughput simulation.

Instead of starting from full precision (FP32) and performing precision reduction, we initialize operator precisions on inference GPUs to the fastest available precision, which minimizes training time while meeting memory constraint $M_i^{max}$. Then we ameliorate the low-precision degradation by recovering some operators to higher-precision formats. A list of heaps $H = \{h_i = heap_{max}(\{[\Omega_o^{(b_{io})} - \Omega_o^{(ADD(b_{io}))}, o_i] \mid \forall o \in O\}) \mid \forall i \in \mathcal{K}_{inf}\}$ is maintained that records the difference of the operator indicator values under the current precision $b_{io}$ and a higher precision[4], together with the operator as the value. The allocator repeatedly checks if operator $o_i$ on an inference GPU can be increased to a higher precision without violating memory and throughput constraints, as precision change can increase memory usage and decrease training speed on the local device; if so, the update is stored and a new candidate is pushed to the max heap if there exists a higher precision for the operator. The process iterates through the candidates in set $H$ and continues until $H$ becomes empty.

To find the optimal initial precision setting that maximizes training throughput, we need to consider the precision of each operator in each local DFG and the casting cost between the operator and its neighbors. An exhaustive search of the entire graph is infeasible due to the high computational complexity. Luckily, many DNN models contain repeating isomorphic building subgraphs [24] which have much fewer precision-adjustable operators available compared with the entire graph. (e.g. BERT's attention has only 5 such operators). We categorize the model into subgraphs and assign a memory budget to each subgraph based on its compression capacity, which is estimated by applying the lowest precision to all operators in the subgraph. A brute-force search is then applied to find the initial precision setting that satisfies local memory constraints while maximizing training speed.

The rationale behind the precision recovery in our allocator design is twofold. Firstly, starting with the highest-performing precision provides a reliable direction for optimization. The shortage for the other case arises from the presence of casting costs, as starting from the highest precision and reducing precision may not always result in faster speed, making it challenging to determine when to stop. Secondly, in practical terms, the precision setting that achieved the highest training throughput is often closer to the optimum. This choice reduces the number of search steps required.

[4]For example, suppose operator $o$ on inference GPU $i$ has three precision candidates, INT8, FP16, and FP32. If $b_{io} = 8$, then higher precision is FP16.

## VI. Backend Optimization and Implementation

A significant challenge in implementing QSync's hybrid-device training lies in the limited support for low-precision kernels in the existing training frameworks [25]. This includes inadequate support for low-precision fixed-point kernels (such as INT8 and INT4), as well as limitations in vendor-optimized black-box kernels in not supporting flexible precision changes (e.g., changing the precision of the output), which loses opportunities for optimizing low-precision kernels for improved device performance [26].

To fully exploit the benefits of low-precision operators on different GPUs, we design and implement LP-PyTorch, a highly *templated* backend that allows kernel configuration to the underlying lower-precision kernels for different operators. LP-PyTorch is designed to use the underlying kernels (e.g., CUTLASS [9] or CuDNN's execution kernels) in a user-friendly and precision-flexible manner. We highlight our two key designs: (1) Multi-Level Abstraction. LP-PyTorch templates each kernel as a combination of hardware-specific configuration and kernel abstractions (e.g. forward and backward pass kernels, tensor precision conversion kernel) to allow maximized flexibility and control over operators's configuration. In practice, we automatically set the composable kernel configuration, such as ThreadblockShape, WarpShape, and InstructionShape, to different precisions to optimize performance on the target hardware platform (such as GPU architecture sm70, sm75, sm80, and simt). (2) Front-end Security Wrapper. The tensorized kernels can have strict requirements for memory access patterns and input data precisions, e.g. TensorCore has restrictions on input tensor dimensions. We wrap kernel calls with security checks and handling using a wrap function.

Several enhancements are included to further reduce overhead and maximize the benefits of low-precision kernels.

**Minmax Optimization.** To calculate the scaling factor for tensor-wise fixed-point quantization, we need to find the maximum and minimum (minimax) values of a tensor. The collection process for large input shapes was observed to suffer from suboptimal GPU utilization. To address this, we developed a GPU kernel to optimize the process. We partition the process into two steps. In the first step, we collected row-wise statistics by evenly partitioning the rows (second-to-last) using a constant number of threads per block. The statistics were obtained through a warp-level primitive. Subsequently, we launched another smaller kernel to the collected row-wise results to obtain the absolute tensor-wise scalar value.

**Dequantization Fusion.** In backpropagation of the low-precision kernels, we output the gradient of weight in FP32, while the gradient of activation maintains FP16 for speed up; also, the fixed-point calculation is done in INT32 and requires additional dequantization before feeding the results into the succeeding operator. To save the dequantization cost, we further fuse the dequantization process into the operator kernel in the epilogue level, i.e., before copying the accumulator result into the shared memory. The lowest computation primitives of CUTLASS are done by tile iterators, QSync specifies a partial iterator method from INT32→FP32, and passes the quantization scaling factor.

## VII. Evaluation

**Testbed.** We evaluate QSync on real-world testbeds. (1) *ClusterA*: a heterogeneous cluster consists of two training servers and two inference servers. Each training server is equipped with eight Nvidia Tesla V100 GPUs with 32GB of memory and 300GB/s interconnect bandwidth. Each inference server is equipped with eight Nvidia T4 GPUs with 16GB of memory and 32GB/s interconnect bandwidth. (2) *ClusterB*: a memory-constrained version of cluster A, where T4 GPUs' available memory is limited to a ratio, we set it as 30% by default, to emulate hybrid training scenarios in a real production system. We use all-reduce for parameter synchronization among GPUs in distributed model training. The software environment includes PyTorch-1.10.0, torchvision-0.11.0 [25] for the convolution-based task, Hugging Face Transformers 4.22.0 [27] for the transformer-based task the and CUDA-11.3.

**Benchmarks.** We mainly evaluate from-scratch training performance for convolution-based models VGG [28] and ResNet [29] for image classification on ImageNet [30]; To show the fidelity of the predictor, we also involve transformer-based finetune task with models BERT [31] on SQuAD [32] for question answering and RoBERTa [33] on SWAG [34] for multiple-choice benchmarks. We choose operator precisions among representative $INT8, FP16$, and $FP32$. Since a bit-width smaller than 16 only supports channels last (NHWC) memory format, for a fair comparison, all convolution-based models are trained under the channels last.

**Training Configurations.** We trained VGG and ResNet models using a local batch size of 128 and a test batch size of 32, along with the SGD optimizer. The learning rate ($lr$) was set to $4.096$ for ResNet and $0.4$ for VGG. Both models underwent training for 120 epochs. For the fine-tuning of RoBERTa, we utilized a local batch size of 16 and a test batch size of 16, employing the Adam optimizer with a learning rate ($lr$) of $7.5e{-}5$. The fine-tuning process lasted for 6 epochs. Similarly, BERT was fine-tuned using a local batch size of 12 and a test batch size of 12, also with the Adam optimizer. The learning rate ($lr$) used was $1.2e-4$, and the training was carried out for 5 epochs.

| model | ClusterA | | ClusterB | |
|---|---|---|---|---|
| | Method | Final Accuracy | Method | Final Accuracy |
| ResNet50 | QSync | **76.77(+0.24)** $\pm$ 0.43% | QSync | **76.67(+0.67)** $\pm$ 0.59% |
| | Random | 76.53 $\pm$ 0.53% | Hess | 76.00 $\pm$ 0.43% |
| VGG16BN | QSync | **74.77(+0.62)** $\pm$ 0.12% | QSync | **74.27(+0.91)** $\pm$ 0.06% |
| | Random | 74.12 $\pm$ 0.88% | Hess | 73.36 $\pm$ 0.63% |
| BERT | QSync | **87.41(+0.02)** $\pm$ 0.05% | QSync | **87.44(+0.10)** $\pm$ 0.20% |
| | Random | 87.39 $\pm$ 0.19% | Hess | 87.34 $\pm$ 0.11% |
| RoBERTa | QSync | 83.59 $\pm$ 0.11% | QSync | **82.94(+0.23)** $\pm$ 0.12% |
| | Random | **83.61(+0.02)** $\pm$ 0.15% | Hess | 82.71 $\pm$ 0.31% |

TABLE II
INDICATOR PERFORMANCE. THE BEST ACCURACY IN EACH SET OF EXPERIMENTS IS MARKED IN BOLD.

**Baselines.** We compare QSync's performance with existing schemes in various aspects: (i) The end-to-end system performance (throughput) and final accuracy with dynamic batch sizing (DBS) [4] and uniform precision (UP), i.e., use a uniform precision for all operators in inference GPU, continue lowering precision until the memory requirement is met; We also compared with an ORACLE accuracy obtained through non-quantized (FP32) training. (ii) Indicator's effect with random and Hessian [8].

**Metrics.** For model accuracy evaluation, we use top-1 accuracy and f1-score for classification and fine-tuning tasks, respectively, and refer to both as accuracy in the results. We evaluate final model accuracy and single-iteration training throughput following study [24], as all our experiments are conducted under the same basic training configurations (such as the total number of epochs, and the learning rate scheduler).

### A. Performance of the Predictor

*1) Indicator Effectiveness:* We compare our indicator with the state-of-the-art Hessian indicator (HESS) method [8] for selecting operators in adaptive fixed-point quantization. HESS computes the block-wise Hessian for each layer and calculates the top eigenvalue, which is then divided by the parameter size and times the introduced error of the quantization. For floating-point quantization, we also compare our indicator with a random scheme. In the later approach, the largest indicator is randomly generated for the lowest precision of each operator and is halved as precision increases. For operators whose fixed-point indicator has been provided by HESS, the floating-point indicator is also halved but take it as a base. To ensure the fairness and clarity of our fixed-point quantization results, we assign different compression ratios for each trial in cluster B. These ratios are determined to emulate a 60% maximum compression level compared to FP32 models.

Table II shows the results of our indicator. In most cases, our indicator achieves higher final model accuracy compared

| Model | Method | Avg. Est. (ms) | Err |
|---|---|---|---|
| Half-Linears | Ground Truth | 474.83 | / |
| | w/o cost mapper(Dpro) | 427.50 | $8 \pm 0.3\%$ |
| | QSync | 474.52 | **3.5 ± 0.5 % (-4.5)** |
| INT-Linears | Ground Truth | 548.46 | / |
| | w/o cost mapper(Dpro) | 462.73 | $13 \pm 1.9\%$ |
| | QSync | 537.55 | **2 ± 0.1 % (-11)** |
| Half-BertLayer1,3,5 | Ground Truth | 787.02 | / |
| | w/o cost mapper(Dpro) | 765.55 | $3 \pm 0.7\%$ |
| | QSync | 781.50 | **1 ± 0.7 % (-2)** |

TABLE III
REPLAY ACCURACY. THE BEST RESULTS ARE MARKED IN BOLD.


Fig. 6. Training timeline of VGG16BN on ClusterA. Top: Uniform precision. Bottom: QSync. QSync recovers accuracy by saving the waiting time.

to existing schemes. We attribute this improved performance on cluster B to the fact that Hessian only considers weight distribution, but does not provide a comprehensive depiction of the negative impact of low-precision kernels on training.

*2) Replay Accuracy:* Table III compares the predicted per-iteration training time with our predictor and DPro [35] against the actual training iteration time measured. We compare the prediction results of BERT when converting all linear layers to half-precision (FP16), int8, and converting three BERT layers to half-precision. We then let Replayer estimate the latency of each of these configurations. Each prediction is repeated 5 times and the average prediction error is calculated. Our system's prediction error is less than 5% in all cases, while Dpro's prediction error is much larger. We attribute it as not considering the casting costs and operator dependency in mixed precision training.

### B. End-to-end Performance of QSync

**Mini Sample of QSync.** Fig. 6 gives the CUDA kernel and communication timeline when training VGG16BN in cluster A using the uniform precision (the top execution timeline) and QSync (the bottom execution timeline). With our precision candidates, uniform precision accelerates computation but leads to workload mismatch, i.e. inference GPU is fully accelerated to be faster than training GPU; then the inference GPUs have to wait until the slow ones finish their tasks before they can start the collective communication and continue execution. QSync recovers some of the FP16 layers to the FP32 format, greatly reducing the waiting time for communication. This improves device utilization while reducing model performance degradation caused by low-precision kernels.

**Performance of QSync.** Table IV presents a comprehensive comparison of training outcomes using QSync, dynamic batch sizing, and uniform precision in the context of ClusterA. In comparison to uniform precision, the adoption of QSync yields superior final accuracy while maintaining consistent throughput. Notably, for the VGG16BN model, an accuracy improvement of up to 0.96% is observed, surpassing even

| Model | Methods | Final Accuracy | Throughput (it/s) |
|---|---|---|---|
| ResNet50 | ORACLE | $76.93 \pm 0.20\%$ | † |
| | DBS | $76.13 \pm 0.05\%$ | 0.40 |
| | UP | $76.50 \pm 0.26\%$ | 0.45 |
| | QSync | **76.77 ± 0.43 % (+0.27)** | **0.45(+0.05)** |
| VGG16 | ORACLE | $70.43 \pm 0.06\%$ | † |
| | DBS | $69.83 \pm 0.15\%$ | 0.17 |
| | UP | $69.76 \pm 0.06\%$ | 0.20 |
| | QSync | **70.33 ± 0.06% (+0.57)** | **0.20(+0.03)** |
| VGG16BN | ORACLE | $74.46 \pm 0.07\%$ | † |
| | DBS | $73.93 \pm 0.15\%$ | 0.32 |
| | UP | $73.80 \pm 0.10\%$ | 0.38 |
| | QSync | **74.77 ± 0.12% (+0.96)** | **0.38(+0.06)** |

TABLE IV
PERFORMANCE OF FROM-SCRATCH TRAINING IN CLUSTERA. IT/S IS THE ITERATION PER SECOND, WHICH MEANS HOW MANY ITERATIONS CAN BE FINISHED WITHIN A SECOND. BEST RESULTS ARE MARKED IN BOLD.

| Model | Methods | Final Accuracy | Throughput (it/s) |
|---|---|---|---|
| ResNet50 | ORACLE | $76.93 \pm 0.20\%$ | † |
| | DBS | $76.40 \pm 0.10\%$ | 0.40 |
| | UP | $76.36 \pm 0.20\%$ | 0.40 |
| | QSync | **76.67 ± 0.59 % (+0.33)** | **0.45(+0.05)** |
| VGG16BN | ORACLE | $74.46 \pm 0.07\%$ | † |
| | DBS | $73.93 \pm 0.15\%$ | 0.32 |
| | UP | $73.23 \pm 0.13\%$ | 0.38 |
| | QSync | **74.26 ± 0.06 % (+1.03)** | **0.38(+0.06)** |

TABLE V
PERFORMANCE OF FROM-SCRATCH TRAINING IN CLUSTERB. BEST RESULTS ARE MARKED IN BOLD.

that attained by single-precision. Furthermore, in terms of throughput, our system consistently achieves a gain of over 10% when compared to dynamic batch sizing across all tasks.

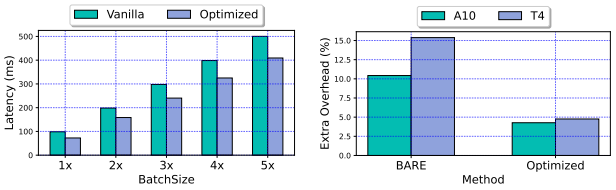
In the context of ClusterB, as demonstrated in Table V, QSync consistently outperforms dynamic batch sizing and uniform precision in terms of model accuracy. Notably, it even achieves a throughput gain compared to uniform precision in the case of ResNet50. This discrepancy becomes more pronounced due to the limited availability of GPU memory, necessitating the adoption of INT8 quantization. It is worth noting that the degradation introduced by INT8 is more severe in comparison to FP16, compounded by the quantization overhead. However, QSync effectively addresses this challenge by recovering unnecessary INT8 operators to their higher precision format, thereby attaining improvements in both accuracy and, remarkably, throughput.

### C. Performance of Transformer-Based Fine-tune Task

As depicted in Table VI, the QSync technique demonstrates consistent speed improvement in quantization while achieving enhanced accuracy when compared to uniform precision. However, it falls short in accuracy compared to dynamic batch sizing. We attribute this discrepancy to the inherent dissimilarities in both the structural aspects and the nature of the tasks involved. Specifically, convolution tasks employ operators that are sensitive to batch size, such as Batch Normalization (BN), whereas transformer tasks utilize Layer Normalization, which is not influenced by batch size variations. Furthermore, it is worth noting that finetuning tasks exhibit less sensitivity to batch size changes in comparison to from-scratch tasks.

| Model | Methods | Final Accuracy | Throughput (it/s) |
|---|---|---|---|
| Bert | ORACLE | $87.49 \pm 0.08\%$ | † |
| | DBS | **87.52** $\pm$ **0.20**% | 1.68 |
| | UP | $87.28 \pm 0.28\%$ | 1.78 |
| | QSync | $87.41 \pm 0.05\%$ **(+0.13)** | **1.78(+0.10)** |
| RoBERTa | ORACLE | $83.95 \pm 0.05\%$ | † |
| | DBS | **83.73** $\pm$ **0.21**% | 1.10 |
| | UP | $83.46 \pm 0.09\%$ | 1.34 |
| | QSync | $83.59 \pm 0.11\%$ **(+0.13)** | **1.34(+0.24)** |

TABLE VI
PERFORMANCE OF FINE-TUNING TASKS IN CLUSTERA.




(a) (b)

Fig. 7. (a) Quantization Overhead Comparison for Fixed Quantization. (b) Extra Overhead Comparison for INT8 With Respect To FP16.

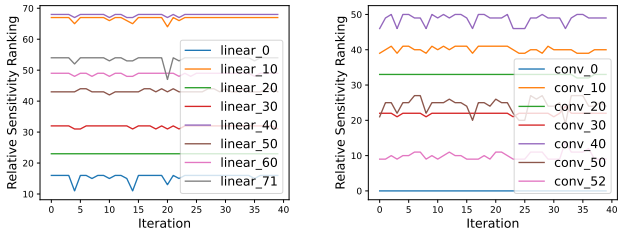
### D. System Optimization

We evaluated to assess the effectiveness of the techniques implemented in LP-PyTorch in enhancing system efficiency. In Fig. 7 (a), we quantified the quantization overhead for a tensor with a shape of $(64, 56, 56)$ and a base batch size of 64, comparing the vanilla implementation of quantization in PyTorch with our optimized approach. We performed five measurements for each method and calculated the average execution cost on the T4 GPU. The results demonstrate a significant overhead reduction of 16-20% in the quantization process, particularly with larger batch sizes.

To further evaluate the impact of the optimization techniques we proposed in LP-PyTorch (calibration optimization and fusion), we compared the additional end-to-end overhead during the training of a ResNet50 model with a batch size of 256 on both the T4 and A10 GPUs using INT8. Fig. 7 (b) illustrates the findings, with the overhead normalized against FP16 training. This experiment was conducted because full INT8 training is typically slower than FP16 due to the cost associated with casting. However, our proposed optimization methods successfully reduce this performance discrepancy from 10% to 5%, indicating improved efficiency for the low-precision fixed-point kernel utilization.

### E. Indicator Trace

We conducted a comprehensive analysis of the indicator variation across multiple layers in two distinct models during the initial 50 updates of the training process. Our empirical findings demonstrate that, while fluctuations were observed between layers, the relative importance and ranking of the layers remained remarkably consistent. Notably, we observed significant disparities in layer sensitivity between the two models. In particular, the layers subsequent to the middle layers, such as the 40th linear or convolution layers, displayed significantly greater sensitivity when compared to the remaining layers.




(a) Bert (b) ResNet50

Fig. 8. Relative Indicator Rank of (a) BERT and (b) ResNet50 for the Initial 50 Training Updates.

## VIII. DISCUSSION AND LIMITATIONS

**Efficient Profiling:** We recognize the considerable profiling overhead associated with the current implementation, which entails running a portion of the training process to trace communication nodes and indicator statistics. To address this, we suggest employing customized communication operations and alternative indicators that are less irrelevant to training progress, enabling more efficient estimation.

**QSync Under Automated Mixed Precision:** AMP employs FP16/BF16 for both inference and training GPUs. We assert QSync is still applicable, with the precision recovery target shifting from the inference GPU to the training GPU. We refer to this scenario as quantization-minimized synchronous training under the throughput-maximum case.

**System Interplay:** It is important to note that adding an inference GPU does not always lead to accelerated training. In our study, we assume that if the inclusion of an inference GPU does not enhance training speed or if the available memory is inadequate, the inference GPU will not be utilized or scheduled. Further investigation into the interplay between system components is left as future work.

## IX. CONCLUSION

We present QSync, a quantization-minimized synchronous training system for hybrid-device training. QSync introduces a Predictor with an Indicator that guides operator precision selection and a Replayer that accurately simulates distributed mixed-precision training. QSync's Allocator interacts with the Predictor to decide efficient low-precision assignments to operators. Implemented on our optimized LP-PyTorch backend, QSync provides access to a wide range of target-tuned low-precision kernels. Through empirical evaluations on various DNN models in real-world training environments, our results demonstrate that QSync effectively mitigates accuracy degradation caused by low-precision operators while maintaining training throughput efficiency.


## ACKNOWLEDGMENT

This work was supported by Hong Kong Innovation and Technology Commission's Innovation and Technology Fund (Partnership Research Programme with ByteDance Limited, Award No. PRP/082/20FX), and grants from Hong Kong RGC under the contracts HKU 17208920 and C7004-22G (CRF).

## REFERENCES


[1] J. H. Park, G. Yun, C. M. Yi, N. T. Nguyen, S. Lee, J. Choi, S. H. Noh, and Y. ri Choi, "HetPipe: Enabling large DNN training on (whimpy) heterogeneous GPU clusters through integration of pipelined model parallelism and data parallelism," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, 2020.

[2] L. Song, F. Chen, Y. Zhuo, X. Qian, H. Li, and Y. Chen, "Accpar: Tensor partitioning for heterogeneous deep learning accelerators," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2020.

[3] J. Li, H.-Y. Xu, Y. Zhu, Z. Liu, C. Guo, and C. Wang, "Aryl: An elastic cluster scheduler for deep learning," *ArXiv*, vol. abs/2202.07896, 2022.

[4] C. Chen, Q. Weng, W. Wang, B. Li, and B. Li, "Semi-dynamic load balancing: efficient distributed learning in non-dedicated environments," *Proceedings of the 11th ACM Symposium on Cloud Computing*, 2020.

[5] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37*, 2015.

[6] P. Goyal, P. Dollár, R. B. Girshick, P. Noordhuis, L. Wesolowski, A. Kyrola, A. Tulloch, Y. Jia, and K. He, "Accurate, large minibatch sgd: Training imagenet in 1 hour," *ArXiv*, vol. abs/1706.02677, 2017.

[7] A. M. Abdelmoniem and M. Canini, "Towards mitigating device heterogeneity in federated learning via adaptive model quantization," in *Proceedings of the 1st Workshop on Machine Learning and Systems*, 2021.

[8] Z. Yao, Z. Dong, Z. Zheng, A. Gholami, J. Yu, E. Tan, L. Wang, Q. Huang, Y. Wang, M. Mahoney *et al.*, "Hawq-v3: Dyadic neural network quantization," in *International Conference on Machine Learning*, 2021.

[9] V. Thakkar, P. Ramani, C. Cecka, A. Shivam, H. Lu, E. Yan, J. Kosaian, M. Hoemmen, H. Wu, A. Kerr, M. Nicely, D. Merrill, D. Blasig, F. Qiao, P. Majcher, P. Springer, M. Hohnerbach, J. Wang, and M. Gupta, "CUTLASS," 2022.

[10] Z. Bai, Z. Zhang, Y. Zhu, and X. Jin, "PipeSwitch: Fast pipelined context switching for deep learning applications," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020.

[11] Y. Wu and J. Johnson, "Rethinking "batch" in batchnorm," *ArXiv*, vol. abs/2105.07576, 2021.

[12] H. Zhang, K. Dana, J. Shi, Z. Zhang, X. Wang, A. Tyagi, and A. Agrawal, "Context encoding for semantic segmentation," in *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018.

[13] NVIDIA, "Nvidia-v100-datasheet," 2018. [Online]. Available: https://images.nvidia.com/content/technologies/volta/pdf/volta-v100-datasheet-update-us-1165301-r5.pdf

[14] ——, "Nvidia-t4-datasheet," 2019. [Online]. Available: https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/tesla-t4/t4-tensor-core-datasheet-951643.pdf

[15] I. Markov, A. Vladu, Q. Guo, and D. Alistarh, "Quantized distributed training of large models with convergence guarantees," *arXiv preprint arXiv:2302.02390*, 2023.

[16] X. Zhang, S. Liu, R. Zhang, C. Liu, D. Huang, S. Zhou, J. Guo, Q. Guo, Z. Du, T. Zhi, and Y. Chen, "Fixed-point back-propagation training," in *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2020.

[17] X. He, J. Sun, H. Chen, and D. Li, "Campo: Cost-Aware performance optimization for Mixed-Precision neural network training," in *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, 2022.

[18] B. Xu, Y. Zhang, H. Lu, Y. Chen, T. Chen, M. Iovine, M.-C. Lee, and Z. Li, "AITemplate," 2022. [Online]. Available: https://github.com/facebookincubator/AITemplate

[19] J. Chen, L. Zheng, Z. Yao, D. Wang, I. Stoica, M. Mahoney, and J. Gonzalez, "Actnn: Reducing training memory footprint via 2-bit activation compressed training," in *International Conference on Machine Learning*, 2021.

[20] D. Basu, D. Data, C. Karakus, and S. Diggavi, "Qsparse-local-sgd: Distributed sgd with quantization, sparsification and local computations," *Advances in Neural Information Processing Systems*, vol. 32, 2019.

[21] F. Fu, Y. Hu, Y. He, J. Jiang, Y. Shao, C. Zhang, and B. Cui, "Don't waste your bits! squeeze activations and gradients for deep neural networks via tinyscript," in *International Conference on Machine Learning*, 2020.

[22] N. Wang, J. Choi, D. Brand, C.-Y. Chen, and K. Gopalakrishnan, "Training deep neural networks with 8-bit floating point numbers," in *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, 2018.

[23] J. H. Lee, S. Ha, S. Choi, W.-J. Lee, and S. Lee, "Quantization for rapid deployment of deep neural networks," *arXiv preprint arXiv:1810.05488*, 2018.

[24] X. Y. Geoffrey, Y. Gao, P. Golikov, and G. Pekhimenko, "Habitat: A {Runtime-Based} computational performance predictor for deep neural network training," in *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, 2021.

[25] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems*, 2019.

[26] J. Xing, L. Wang, S. Zhang, J. Chen, A. Chen, and Y. Zhu, "Bolt: Bridging the gap between auto-tuners and hardware-native performance," in *Proceedings of Machine Learning and Systems*, 2022, pp. 204–216.

[27] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz, J. Davison, S. Shleifer, P. von Platen, C. Ma, Y. Jernite, J. Plu, C. Xu, T. L. Scao, S. Gugger, M. Drame, Q. Lhoest, and A. M. Rush, "Transformers: State-of-the-art natural language processing," in *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, 2020.

[28] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *International Conference on Learning Representations*, 2015.

[29] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[30] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *2009 IEEE Conference on Computer Vision and Pattern Recognition*, 2009, pp. 248–255.

[31] J. D. M.-W. C. Kenton and L. K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," in *Proceedings of naacL-HLT*, 2019.

[32] P. Rajpurkar, J. Zhang, K. Lopyrev, and P. Liang, "Squad: 100,000+ questions for machine comprehension of text," *arXiv preprint arXiv:1606.05250*, 2016.

[33] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, "Roberta: A robustly optimized bert pretraining approach," *arXiv preprint arXiv:1907.11692*, 2019.

[34] R. Zellers, Y. Bisk, R. Schwartz, and Y. Choi, "SWAG: A large-scale adversarial dataset for grounded commonsense inference," in *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, 2018.

[35] H. Hu, C. Jiang, Y. Zhong, Y. Peng, C. Wu, Y. Zhu, H. Lin, and C. Guo, "dpro: A generic performance diagnosis and optimization toolkit for expediting distributed dnn training," in *Proceedings of Machine Learning and Systems*, 2022.