Micro Batch Streaming: Allowing the Training of DNN models Using a large batch size on Small Memory Systems
DoangJoo Synn, XinYu Piao, JooYoung Park and Jong-Kook Kim
Department of Electrical and Computer Engineering, Korea University
{alansynn, xypiao97, nehalem, jongkook} @korea.ac.kr

Abstract—The size of the deep learning models has greatly increased over the past decade. Such models are difficult to train using a large batch size, because commodity machines do not have enough memory to accommodate both the model and a large data size. The batch size is one of the hyper-parameters used in the training model, and it is dependent on and is limited by the target machine memory capacity and it is dependent on the remaining memory after the model is uploaded. A smaller batch size usually results in performance degradation. This paper proposes a framework called Micro-Batch Streaming (MBS) to address this problem. This method helps deep learning models to train by providing a batch streaming algorithm that splits a batch into the appropriate size for the remaining memory size and streams them sequentially to the target machine. A loss normalization algorithm based on the gradient accumulation is used to maintain the performance. The purpose of our method is to allow deep learning models to train using mathematically determined optimal batch sizes that cannot fit into the memory of a target system.

I. INTRODUCTION

Machine learning methods using Deep Neural Network (DNN) models need to carefully calibrate hyper-parameters to obtain optimal performance. Hyper-parameters are elements of an overall configuration external to the learning model and are used in the learning process to determine model parameter values during model training. The mini-batch size is a crucial hyper-parameter that determines the size or number of the training set in each iteration, and mini-batch size affects the overall model performance, such as accuracy and training time. A mathematically optimal mini-batch size for a particular training model can be determined and it is usually larger than mini-batch sizes commonly used [1]. Mini-batch training is a method that aggregates multiple examples at each iteration of the full batch [figure 1]. Mini-batch size affects the model accuracy in several models such as self-supervised models, and figure 2 shows the effect of different mini-batch size on accuracy [9] for training the ResNet-50 [3] model. As the mini-batch size increases, the performance increases.

The number of parameters of deep learning models has increased exponentially during the last decade, as shown in Figure [1]. Thus, requiring more and more memory for deep learning. Therefore, models to use large batches for training on commodity machines with small memory becomes impossible or very difficult due to lack of memory capacity, not from the lack of computational power. Modern machine learning frameworks such as PyTorch [4], and Tensorflow [5] provide rich features for machine learning researchers to define the appropriate mini-batch size. However, the mini-batch size cannot be increased beyond a certain level due to limitations of the free memory space of the target device because usually memory is pre-allocated for the model. If the mini-batch size is larger than the free memory size, the mini-batch cannot be allocated to the GPU memory and thus the model cannot be trained.

The data size of training examples such as image resolution affects the total data size and the increase in sample data size increases the amount of machine memory required for a mini-
Fig. 3: The deep learning model size growth for image classification using ImageNet dataset.

batch. For example, even with the same mini-batch size, the total data size of the ImageNet dataset is larger than that of the CIFAR-10. Because the image size of the ImageNet dataset is 227 by 227, whereas the image size of the CIFAR-10 dataset is 32 by 32. Therefore, the significant increase in image size makes it more challenging to train models.

Many researchers employ data parallelism and/or model parallelism for problems that deep learning methods face. Data parallelism (e.g., [19], [20], [21]) is usually used when the batch size is too large to be fit into a single device’s memory. A batch is partitioned for computation and sent to multiple devices and each device will have a full copy of the learning model. When all data within a batch is processed, weights are updated across devices through communications. Model parallelism is usually used when the model is too large to fit into a device’s memory (e.g., [13] [17] [18]). This method partitions the learning model into cells and distributes the cells to multiple devices. These methods still has the problem of the mini-batch size being limited by the free memory size.

Another method called pipeline parallelism is proposed (e.g., [7], [10]). These methods employ Data-Parallel Synchronous Stochastic Gradient Descent (SGD), which distributes mini-batches across many machines and pipeline executes them. Our method extends the micro-batch that is mentioned in these researches and the idea of pipeline parallelism is used.

In this paper, a novel framework called Micro-Batch Streaming (MBS) is proposed. MBS is a method that can fetch a large batch of data using the stream-based pipeline scheme to train models even if the batch cannot fit into the memory. The idea is to split a mini-batch into n micro-batches and stream them sequentially to a GPU. The micro-batch in [7], [10] is used and extended for this research.

Results show that MBS can increase the training batch size to the full size of the training set regardless of the type of models, datasets, and data size. To maintain the performance, MBS computes the gradient for a large batch using loss normalization based on micro-batch gradient accumulation, a method that accumulates the gradient of multiple micro-batches.

This paper is organized as follows. Related work is described in Section 2. Section 3 introduces MBS and terminologies used throughout this paper. The proof of concept about the loss normalization algorithm is presented in Section 4. Section 5 depicts the results. Finally, Section 6 concludes the paper.

II. RELATED WORK

If model parallelism is done naively, GPUs suffer from idle time overhead because a worker have to wait for previous or successor worker to finish its job. To minimize this overhead, GPipe [7], a model-parallelism library for training large neural networks, presented a novel pipeline parallelism with batch splitting. GPipe splits mini-batch into smaller batches, called micro-batch, for pipeline execution across multiple GPUs. This enables each GPU to work on different micro-batch simultaneously, and gradient and model parameters are updated synchronously after all micro-batches are processed. However, synchronizing at the end of every mini-batch still causes GPU stalls. Pipedream [10] used asynchronous communication method and corresponding work scheduling and learning method to handle this problem. But neither of these solutions are adequate for large mini-batch training because they focus on maximizing the learning performance. and they have memory limitation where the batch size cannot exceed the free memory space of a GPU. Our method solves this limitation.

III. MICRO-BATCH STREAMING

Fig. 4: Overview of Micro-Batch Streaming.

Micro-Batch Streaming (MBS) is a novel method based on stream-based pipeline computation to train models using large batches and is also well-suited with both data parallelism and model parallelism paradigm. This section describes our proposed method and its major characteristics.

A. Stream-based Pipeline

The stream-based pipeline (SP) is a pipeline method to split data and stream data sequentially to target machines. The batch data is pre-loaded into the main memory before being loaded
and trained in the GPU. Usually the memory size of the target device, such as a GPU, is smaller than the CPU memory size. Thus, it may not be possible to load a large batch of data into the GPU memory after data is loaded into CPU memory for training. The stream-based pipeline used in MBS splits a large batch of data into smaller batches that can be allocated to the GPU memory. Therefore, if it is required, the GPU can compute the entire batch by training small batches streamed from the CPU.

### B. Micro-Batch

In MBS, the micro-batch in [7] is extended and is defined as: 1) a unit of streaming to the target machine, and 2) a unit of computing on the target machine. The first definition presents the splitting of the input mini-batch into small batches that fit on a small memory space and streams them sequentially to the target machine. The second definition describes that the target machine trains the neural network using this unit size. The formulation of the micro-batch can be expressed as follows.

\[
\text{mini batch} = \bigcup_{i \in C} \mu\text{batch}_i
\]

\[
\mu\text{batch}_i \subset \text{mini batch} \quad (i \in C)
\]

\[
\mu\text{batch}_i \subset \text{mini batch} \quad (i \in C)
\]

The micro-batch effectively removes the dependency between the batch size and the target device’s memory capacity. The mini-batch can be used as a milestone to update accumulated gradient to the model’s parameters and its magnitude can be scaled by hyper-parameters such as the learning rate. On the other hand, the micro-batch is not affected by hyper-parameters because it is a unit of streaming and computation on the GPU. Therefore, a learning model can be trained as if using a large mini-batch size, although the target device is trained using micro-batches.

### C. Process flow of MBS

Figure 4 shows the overview of the process flow of MBS between CPU and GPU. The input batch is split into micro-batches in the CPU, and these micro-batches are sequentially streamed by the GPU to be computed in GPU. Before describing the processing flow of MBS between CPU and GPU, we need to understand how GPU memory is used while training the model. When the model starts to train, the GPU memory is split into two memory domains (shown in figure 5); the model parameters and gradient memory space (model parameter space), and the other is the dataset and intermediate outcomes memory space (dataset space).

The data in the model parameter space is used to determine the updates of the model’s parameters. The intermediate outcomes in the data set space that are used to calculate the gradient later, are computed by the forward pass in the model and are completely different values compared to the gradient.

The process of MBS training a model between CPU and GPU is as follows (figure 5). First, MBS loads the mini-batch dataset to the CPU memory space and then splits the mini-batch into \( n \) micro-batches to prepare to stream to GPU. Then, MBS streams micro-batches sequentially to the GPU, and the GPU starts to train the model using each micro-batch datasets. The GPU executes the forward and backward steps and stores the gradient to the model parameter space. Since not all the micro-batches in the mini-batch is completed, the model parameters are not updated and GPU computes the next micro-batch. The the gradients computed by the forward and backward propagation are accumulated until all micro-batch is processed. When the final micro-batch is processed, MBS updates the model parameters using the accumulated gradient, similar to the gradient computed by the mini-batch. Therefore, it is as though the model’s parameters are updated using the mini-batch, although the gradients are computed from micro-batches.

### D. Loss Normalization

The accumulated gradient from subsection III.B. is not equal to the gradient computed by using a mini-batch. Therefore, this paper introduces a loss normalization method that normalizes the loss computed by a loss function at every micro-batch iteration. The loss normalization method provides the same effect of normalizing the gradient. The reason for presenting and using the loss normalization method rather than the gradient normalization method is that the type of the loss value is a scalar-tensor. Scalar-tensor type has an advantage in computation time. The necessity and appropriateness of loss normalization is described in detail in section IV. In MBS, the loss normalization method is used for the automatic update of gradients and MBS waits until all micro-batches, split from a mini-batch, are completed before the update. Thus it will be the same as an update after a mini-batch is completed.

### IV. PROOF OF CONCEPT

Stochastic Gradient Descent (SGD) [26] is one of the widely used algorithms for deep learning. SGD computes the loss from the difference between outputs computed from the forward pass in the model and targets. However, if the calculation of loss and the gradient accumulation are not considered for MBS, the model may not be trained correctly as intended. With loss accumulation without normalization, the significance of all datasets by loss computation for each micro-batch does not match but accumulates as an immense, unscaled value. Therefore, we introduce the loss normalization method to scale the loss across the mini-batch. This section defines the problem in gradient accumulation and proves the necessity and appropriateness of the loss normalization method introduced in MBS.

#### A. Problem in the gradient accumulation

Gradient accumulation concentrates gradients at every micro-batch iteration. Although the gradient accumulation can store accumulated gradients from iterations, it does not
Fig. 5: Flow diagram of MBS. (1) MBS splits the input batch into a set of micro-batches. (2) MBS streams micro-batches sequentially to the GPU. (3) The GPU computes and accumulates the gradient. (4) Repeats this process until the final micro-batch is finished. (5) When the final micro-batch is finished, the model is updated using the accumulated gradient.

Fig. 6: Illustration of updating the model’s parameter when the final micro-batch for a mini-batch is completed.

Consider a large batch size to calculate the gradient nor normalize the gradients to the corresponding size. Hence, it only calculates the sum of the previous gradient and the current gradient. For instance, most loss functions, such as mean square error, compute the value of loss for the current input tensor size only, and do not consider the total size of a large mini-batch where current input tensor belongs.

Therefore, the accumulated gradient and the gradient of a mini-batch are not equal since gradient accumulation does not normalize the accumulated gradient to the gradient of a mini-batch.

$n$ in equation [1] means the size of the input batch that is entered a loss function. Equation [6] shows that gradients accumulate using the gradient accumulation. However, the result of equation [6] is not equal to the gradient of a mini-batch (equation [4]).

\[
\text{loss} = \frac{1}{n} \sum_{i=1}^{n} (\text{output}_i - \text{target}_i)^2 \quad (1)
\]

\[
\text{loss} = \frac{1}{n} \sum_{i=1}^{n} A_i \quad (2)
\]

\[
\nabla W = \nabla W_{mb} = \partial \left( \frac{1}{n} \sum_{i=1}^{n} A_i \right) \quad (3)
\]

\[
\nabla W_{\mu} = \partial \left( \frac{1}{n} \sum_{i=1}^{n} A_i \right) \quad (4)
\]

\[
\text{grad}_{\text{accum}} = \sum_{k=1}^{c} \partial (\text{loss}_k) \quad (5)
\]

\[
\text{grad}_{\text{accum}} = \sum_{k=1}^{c} \partial \left( \frac{1}{n} \sum_{i=1}^{n} A_i \right) \quad (6)
\]

Consider $n$ the size of the mini-batch, $m$ the size of the micro-batch, and $c$ the number of subsets of the micro-batch, as follows:

\[
c = \frac{n}{m} \quad (8)
\]

\[
\nabla W_{mb} = \partial \left( \frac{1}{n} \sum_{i=1}^{n} A_i \right) \quad (9)
\]

\[
\nabla W_{\mu} = \partial \left( \frac{1}{m} \sum_{j=1}^{m} A_j \right) \quad (10)
\]
\( \nabla W_{mb} \) means the gradient of a mini-batch, and \( \nabla W_{m} \) means the gradient of a micro-batch in equation 9 and 10. Consequently, equation 9 and equation 6 are not equivalent.

The below equations prove that the accumulated gradient is not equal to the gradient of a mini-batch. Furthermore, we proved that there is a need for a method to normalize the gradients of a mini-batch.

\[
\sum_{i=1}^{n} A_i = \sum_{j=1}^{c} \sum_{k=1}^{m} A_{j,k} \quad (11)
\]

\[
\nabla W_{mb} = \partial \left( \frac{1}{n} \sum_{i=1}^{n} A_i \right) \quad (12)
\]

\[
= \partial \left( \frac{1}{n} \sum_{j=1}^{c} \sum_{k=1}^{m} A_{j,k} \right) \quad (13)
\]

\[
= \partial \left( \frac{1}{c \cdot m} \sum_{j=1}^{c} \sum_{k=1}^{m} A_{j,k} \right) \quad (14)
\]

\[
= \partial \left( \frac{1}{c} \sum_{k=1}^{m} (1/m) \sum_{j=1}^{c} A_{j,k} \right) \quad (15)
\]

\[
= \partial \left( \frac{1}{c} \sum_{k=1}^{m} (loss_{\mu})_{k} \right) \quad (16)
\]

\[
\neq \text{grad accum} \quad (17)
\]

\[
= \sum_{k=1}^{c} \partial ((loss_{\mu})_{k}) \quad (18)
\]

\[
= \partial (\sum_{k=1}^{c} (loss_{\mu})_{k}) \quad (19)
\]

### B. Loss Normalization

Loss normalization is a computation method to calculate the gradient of a mini-batch and calculates in the gradient accumulation. The formulation for loss normalization is described as follows.

\[
loss_{\text{norm}} = \frac{1}{c} \sum_{k=1}^{c} (loss_{\mu})_{k} \quad (20)
\]

\[
= \frac{1}{c} \sum_{k=1}^{c} \left( \frac{1}{m} \sum_{j=1}^{m} A_{j,k} \right) \quad (21)
\]

The gradient normalized by the loss normalization is accumulated at every iteration in MBS. However, the mini-batch size is not guaranteed to be uniform for all iterations. If mini-batch size determined by hyper-parameter does not divide the number of total training examples into uniform, the last input batch size is not equal to the other input batch size. In other cases, if the last input batch is smaller than the micro-batch size, the gradient is miscalculated. Therefore, loss normalization dynamically determines the normalizing factor and normalizes the loss considering the current input batch size. The normalizing factor is the number of subsets of micro-batches, and the normalizing factor is set to 1, if the last input batch size is smaller than the micro-batch size. The overall loss normalization algorithm is as follows:

#### Algorithm 1 Loss Normalization

**Require:** a mini-batch \( M \)

\[
\mu B = \text{micro-batches split from } M
\]

\[
c = \text{number of } \mu B
\]

\[
\text{counter} = \text{MBS counter to check iteration}
\]

\[
\text{while} \ counter + 1 < c \ do
\]

\[
\text{loss} \leftarrow \text{fn(output, target)}
\]

\[
\text{loss} \leftarrow \text{loss} / \mu
\]

\[
\text{back-propagation for loss}
\]

\[
\text{counter} \leftarrow \text{counter} + 1
\]

\[
\text{end while}
\]

To calculate the gradient of mini-batch, we considered two normalization methods; 1) the gradient normalization, 2) the loss normalization. These two methods provide the same effect of normalizing the gradient. The gradient normalization method computes using the gradient calculated by each micro-batch, whereas the loss normalization method computes using the loss. However, gradient normalization uses a complex algorithm to normalize the accumulated gradients, and it causes overhead in run-time. On the other hand, the loss normalization does not have an overhead in run-time because it can easily normalize by dividing loss. Therefore, we introduce the loss normalization to normalize the gradient because this method provides the same effect of normalizing, and also no additional computation is required.

#### C. Batch Normalization with Micro-Batch

In deep learning, model includes batch normalization layers to avoid gradient vanishing, exploding or internal covariance shift \( [9] \). Batch normalization manipulates batch statistics to prevent gradient from vanishing. Therefore, training time is reduced by the normalized gradients. However, batch normalization has a strong dependency with input batch dimensions equal to the mini-batch size.

#### Algorithm 2 Batch Normalization via Mini-Batch

\[
\mu B \leftarrow \frac{1}{n} \sum_{i=0}^{n} x_i
\]

\[
\delta_B^2 \leftarrow \frac{1}{n} \sum_{i=0}^{n} (x_i - \mu_i)^2
\]

\[
\hat{x}_i \leftarrow \frac{x_i - \mu_B}{\sqrt{\delta_B^2 + \varepsilon}}
\]

\[
y_i \leftarrow \gamma \hat{x}_i + \beta \equiv BN_{\gamma,\beta}(x_i)
\]

Batch normalization computes statistics in forward pass and both mean and variance are calculated by the input size. However, the calculation of batch normalization statistics in MBS needs to wait until the final micro-batch is processed. This characteristic makes batch normalization computes and updates new statistics by the following equation for every micro-batch. Thus, the calculated statistics are not equal to the statistics of a mini-batch.
\[ BN_{new} = (1 - \text{momentum}) \times \hat{BN} + \text{momentum} \times BN_{new} \] 

In equation (22), \( \hat{BN} \) is the estimated statistic, and \( BN_{new} \) is the new observed value. Tensors that are normalized by batch normalization affect the results of back-propagation. Therefore, this paper defines and implements an accumulation batch normalization appropriate to the gradient accumulation method to calculate statistics as if using mini-batch. The accumulation batch normalization continues to accumulate the sum of tensor size and tensors until the accumulated sum of tensor size is equal to the size of a mini-batch. Then, the accumulation batch normalization computes and updates new statistics as if calculating by a mini-batch. Therefore, the accumulation batch normalization is crucial for calculating the mini-batch statistics through the micro-batch processed sequentially.

The accumulation batch normalization computes through the following equations.

\[
\text{accum sum} = \sum_{i=0}^{c} \left( \sum_{j=0}^{m} x_{i,j} \right)
\]  

(23)

\[
\text{accum square sum} = \sum_{i=0}^{c} \left( \sum_{j=0}^{m} x_{i,j}^2 \right)
\]  

(24)

\[
\hat{BN}_{new} = 1 \times \hat{BN} + 0 \times BN_{new}
\]  

(25)

First, it accumulates the sum of input tensors, the sum of square of input tensors, and the sum of size of input tensors. Until the accumulated sum of tensor size is equal to the size of a mini-batch, accumulation batch normalization does not compute new statistics using zero momentum as equation (25). This is because batch normalization for each micro-batch should not be applied.

\[
\hat{\mu}_B = \frac{1}{c \times m} \left( \sum_{i=0}^{c} \left( \sum_{j=0}^{m} x_{i,j} \right) \right)
\]  

(26)

\[
\hat{\sigma}_B^2 = \frac{1}{c \times m} \left( \sum_{i=0}^{c} \left( \sum_{j=0}^{m} x_{i,j}^2 \right) \right) - \hat{\mu}_B^2
\]  

(27)

\[
BN_{new} = (1 - \text{momentum}) \times \hat{BN} + \text{momentum} \times BN_{new}
\]  

(28)

When all micro-batch is passed, the accumulation batch normalization computes mean and variance like equation (26) and equation (27). In equation (26), \( c \) is number of subsets of micro-batch and \( m \) is the size of a micro-batch. The result of \( c \) times \( m \) is equal to a mini-batch size. Therefore, this mean in equation (26) is equal to the mean for a mini-batch, the variance also. Then, the accumulation batch normalization computes and updates the new batch normalization using these mean and variance like equation (28).

V. EVALUATION

A. Experimental Setup

CIFAR-10 and CIFAR-100 datasets are used to evaluate our proposed method. CIFAR-10 dataset consists of 60,000 32 × 32 colour images in 10 classes, with 6,000 images per class. CIFAR-100 dataset is the same number as CIFAR-10 except for the number of classes which is 100 and each class contains 600 images. These datasets consist of 50,000 training example images and 10,000 test images.

Three models (VGG-16 model [12], ResNet-50, and ResNet-152 [3]) are used for image classification to evaluate our proposed method. The batch normalization layers, which compute the statistics along the input mini-batch dimension, are included in ResNet and VGG models. For MBS, the GPU computes statistics along the micro-batch dimension leading to incorrect model training. Therefore, MBS uses the pipeline batch normalization method proposed by GPipe to address this problem [7].

During training, pre-trained models and the learning rate warm-up techniques such as learning rate scheduler are not used. Because the learning rate scheduler controls the learning rate and is used to avoid falling into the local minima, and only affects when updating model parameters using the gradient, does not affect the gradient calculation. MBS uses the default training algorithm in PyTorch to evaluate performance. ResNet models (ResNet-50, ResNet-152) are trained using the SGD optimizer (0.1 learning-rate, 0.9 momentum, 0.0005 decay). VGG-16 model is trained using the SGD optimizer (0.05 learning-rate, 0.9 momentum, 0.0005 decay).

The experiments are run on a system consisting of GeForce RTX 2080 Ti GPU with 11GB GDDR6 memory, AMD Ryzen Threadripper 3970X processor, and 128GB main memory. MBS was implemented by using PyTorch version 1.9.0 and CUDA version 11.1.

B. Experimental Results

Three cases are compared; 1) the baseline is a mini-batch training method, 2) MBS is our proposed method that does not consider batch normalization (BN) layers, and 3) MBS-BN is our proposed method that includes batch normalization layers.

1) Performance: In this section, we show that MBS provides performance gain in models when using the size of mini-batch or micro-batch shown in table [1]. In this experiment, the micro-batch size is experimentally determined to be a quarter of the mini-batch size for both MBS and MBS-BN.

| Model   | Baseline Mini-batch Size | MBS Micro-batch Size |
|---------|--------------------------|----------------------|
| ResNet-50 | 256                      | 64                   |
| ResNet-152 | 128                     | 32                   |
| VGG-16   | 512                      | 128                  |

TABLE I: The size of mini-batch is the maximum size that can be allocated to the GPU memory and the size of micro-batch is a quarter of the mini-batch size.
Figure 7 shows the performance of each model. In this result, MBS provides accuracy gain compared to its baseline as shown in Table II.

Table II: Results show the maximum accuracy for each case. MBS has the highest accuracy than other results.

| Dataset  | Model    | Baseline | MBS-BN | MBS  |
|----------|----------|----------|--------|------|
| CIFAR-10 | ResNet-50| 86.95    | 87.26  | 88.23|
|          | ResNet-152| 84.56    | 83.04  | 86.60|
|          | VGG-16   | 88.45    | 88.12  | 89.07|
| CIFAR-100| ResNet-50 | 59.01    | 59.72  | 61.74|
|          | ResNet-152| 56.75    | 56.48  | 60.54|
|          | VGG-16   | 59.41    | 58.04  | 61.22|

Table II shows the highest accuracy of each model in more detail. The accuracy of MBS is increased by a maximum of 3.79% than the baseline and is increased by a maximum of 4.05% than MBS-BN. As described in section 4.C, statistics of batch normalization is computed in the forward pass. MBS computes the statistics of batch normalization from every micro-batch; therefore, the number of calculations of batch normalization statistics is larger than the baseline. The batch normalization calculates and updates new statistics using equation 22 and normalizes tensors frequently. These normalized tensors affect the loss normalization results, and then the loss normalization updates the model’s parameters. Therefore, MBS shows the highest accuracy among MBS-BN and baseline. Baseline and MBS-BN are similar because MBS-BN is equal to the baseline in computation frequency and in the statistics update of batch normalization.

2) Maximum mini-batch size: The results in Table III and Table IV show the maximum mini-batch size that each method can train using 32 × 32 image sizes. Without MBS, ResNet-50 model can train up to 256 mini-batch size, and ResNet-152 can train up to 128 mini-batch size. This is because ResNet-152 model has more parameters than ResNet-50 and thus, ResNet-152 cannot allocate a larger mini-batch size than 128. On the other hand, the models with MBS can train large mini-batch sizes without limit. In the results of VGG-16, VGG-16 can train up to 2048 mini-batch size without MBS, whereas using MBS can train large mini-batch sizes without limit.

| Dataset  | Model    | Baseline | MBS  |
|----------|----------|----------|------|
| CIFAR-10 | ResNet-50| ✓        | ✓    |
|          | ResNet-152| ✓        | ✓    |
|          | VGG-16   | ✓        | ✓    |

| Dataset  | Model    | Baseline | MBS  |
|----------|----------|----------|------|
| CIFAR-10 | ResNet-50| ✓        | ✓    |
|          | ResNet-152| ✓        | ✓    |
|          | VGG-16   | ✓        | ✓    |

| Dataset  | Model    | Baseline | MBS  |
|----------|----------|----------|------|
| CIFAR-10 | ResNet-50| ✓        | ✓    |
|          | ResNet-152| ✓        | ✓    |
|          | VGG-16   | ✓        | ✓    |

Table III: Comparison of the maximum mini-batch size of both baseline and MBS in ResNet models using 32 × 32 default image size.

| Model   | Dataset    | Baseline | MBS  |
|---------|------------|----------|------|
| VGG-16  | CIFAR-10   | ✓        | ✓    |
|         | CIFAR-100  | ✓        | ✓    |

Table IV: Comparison of the maximum mini-batch size of both baseline and MBS in VGG-16 using 32 × 32 default image size.
The next experiment is to determine how many mini-batch can be allocated after increasing image size like ImageNet image size. As image size increases, the data size also increases. Therefore, the mini-batch with a large image size is difficult to allocate to GPU memory even if this mini-batch size is equal to the mini-batch size with small image sizes.

| Dataset   | Model    | Baseline | MBS |
|-----------|----------|----------|-----|
| CIFAR-10  | ResNet-50 | ✓  ✓  ✓  ✓ | ✓  ✓  ✓  ✓ |
| CIFAR-10  | ResNet-152 | ✓  ✓  ✓  ✓ | ✓  ✓  ✓  ✓ |
| CIFAR-100 | ResNet-50 | ✓  ✓  ✓  ✓ | ✓  ✓  ✓  ✓ |
| CIFAR-100 | ResNet-152 | ✓  ✓  ✓  ✓ | ✓  ✓  ✓  ✓ |
| CIFAR-100 | ResNet-50 | ✓  ✓  ✓  ✓ | ✓  ✓  ✓  ✓ |
| CIFAR-100 | ResNet-152 | ✓  ✓  ✓  ✓ | ✓  ✓  ✓  ✓ |

TABLE V: Comparison of the maximum mini-batch size of both baseline and MBS in ResNet models using 227 × 227 increased image size.

In Table V, ResNet-50 can allocate up to 4 mini-batch, and ResNet-152 can allocate up to 2 mini-batches. VGG-16 can allocate up to 32 mini-batches shown in Table VI. Results show that MBS allows a large number of mini-batches to be allocated up to 4096 mini-batches which is 1000 times more than the baseline for ResNet models and more than 100 times more than the baseline for VGG-16. This is the effect of MBS streaming micro-batches to the GPU memory.

3) Acceleration in training time: MBS also provides gain in training time. Table VII shows the results of average epoch times for each case. For both ResNet-50 and VGG-16 models, the results show that MBS is fastest. Micro-batch transfers faster than mini-batch because the data size of micro-batch is smaller than mini-batch, which boosts training speed. Figure 8(a) is an illustrated diagram for compute process along deep learning models. Figure 8(b) and 8(c) are illustrated diagrams for training speed gain using micro-batch.

Moreover, the training time of MBS-BN is larger than the baseline. Because batch normalization in MBS has an additional computation cost in the forward pass, models with MBS-BN spend more time compared to other cases. In addition, as batch normalization layers increase, the number of accumulation batch normalization also increases and causes more extra training time.

VI. CONCLUSION

This paper proposes and introduces MBS (Micro-Batch Streaming), a novel method based on stream-based pipeline computation to train models with large batches on small memory size machines. The loss normalization provides a method to minimize the performance loss of training using a small batch size. Compared to neural network models with batch size limitations because of the machine’s memory size, MBS can emulate the training using a large batch up to the maximum number of training samples without performance decrease.

ACKNOWLEDGMENT

This research was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education (NRF-2016R1D1A1B04933156). This work was supported in part by the National Research Foundation of Korea (NRF) through the Basic Science Research Program funded by the Ministry of Education under Grant 2014R1A1A2059527, and in part by the Information Technology Research Center (ITRC), Ministry of Science and ICT (MSIT), South Korea, through a Support Program under Grant IITP-2020-01433, supervised by the Institute for Information and Communications Technology Promotion (IITP)

REFERENCES

[1] Yang You, Zhao Zhang, Cho-Jui Hsieh, James Demmel, and Kurt Keutzer. 2018. "ImageNet Training in Minutes." In Proceedings of the 47th International Conference on Parallel Processing (ICPP 2018).
[2] You, Yang, Igor Gitman, and Boris Ginsburg. "Large batch training of convolutional networks." arXiv preprint arXiv:1708.03888 (2017).
[3] He, Kaiming, et al. "Deep residual learning for image recognition." Proceedings of the IEEE conference on computer vision and pattern recognition. 2016.
[4] Paszke, Adam, et al. "Pytorch: An imperative style, high-performance deep learning library." Advances in neural information processing systems 32 (2019): 8026-8037.
[5] Abadi, Martin, et al. "Tensorflow: A system for large-scale machine learning." 12th USENIX symposium on operating systems design and implementation (OSDI 16). 2016.
[6] Deng, Jia, et al. "Imagenet: A large-scale hierarchical image database." 2009 IEEE conference on computer vision and pattern recognition. Ieee, 2009.

[7] Huang, Yanping, et al. "Gpipe: Efficient training of giant neural networks using pipeline parallelism." Advances in neural information processing systems 32 (2019): 103-112.

[8] Chen, Ting, et al. "A simple framework for contrastive learning of visual representations." International conference on machine learning. PMLR, 2020.

[9] Ioffe, Sergey, and Christian Szegedy. "Batch normalization: Accelerating deep network training by reducing internal covariate shift." International conference on machine learning. PMLR, 2015.

[10] Narayanan, Deepak, et al. "PipeDream: generalized pipeline parallelism for DNN training." Proceedings of the 27th ACM Symposium on Operating Systems Principles. 2019.

[11] Li, Youjie, et al. "Doing more with less: training large DNN models on commodity servers for the masses." Proceedings of the Workshop on Hot Topics in Operating Systems. 2021.

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

[13] Dean, Jeffrey, et al. "Large scale distributed deep networks." Advances in neural information processing systems 25 (2012): 1223-1231.

[14] Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E. Hinton. "Imagenet classification with deep convolutional neural networks." Advances in neural information processing systems 25 (2012): 1097-1105.

[15] Krizhevsky, Alex. "One weird trick for parallelizing convolutional neural networks." arXiv preprint arXiv:1404.5997 (2014).

[16] Jia, Zhihao, et al. "Exploring hidden dimensions in parallelizing convolutional neural networks." arXiv preprint arXiv:1802.04924 (2018).

[17] Lee, Seunghak, et al. "On model parallelization and scheduling strategies for distributed machine learning." (2014).

[18] Ben-Nun, Tal, and Torsten Hoefler. "Demystifying parallel and distributed deep learning: An in-depth concurrency analysis." ACM Computing Surveys (CSUR) 52.4 (2019): 1-43.

[19] Jia, Zhihao, Matei Zaharia, and Alex Aiken. "Beyond data and model parallelism for deep neural networks." arXiv preprint arXiv:1807.05358 (2018).

[20] Ho, Qirong, et al. "More effective distributed ml via a stale synchronous parallel parameter server." Advances in neural information processing systems. 2013.

[21] Huo, Zhouyuan, Bin Gu, and Heng Huang. "Decoupled parallel backpropagation with convergence guarantee." International Conference on Machine Learning. PMLR, 2018.

[22] Goyal, Priya, et al. "Accurate, large minibatch sgd: Training imagenet in 1 hour." arXiv preprint arXiv:1706.02677 (2017).

[23] Keskar, Nitish Shirish, et al. "On large-batch training for deep learning: Generalization gap and sharp minima." arXiv preprint arXiv:1609.04836 (2016).

[24] Li, Mu, et al. "Efficient mini-batch training for stochastic optimization." Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining. 2014.

[25] Chen, Jianmin, et al. "Revisiting distributed synchronous SGD." arXiv preprint arXiv:1604.00981 (2016).

[26] J. Kiefer and J. Wolfowitz Stochastic Estimation of the Maximum of a Regression Function Ann. Math. Statist. Volume 23, Number 3 (1952), 462-466.