Improvements in GPipe Pipeline Parallel Acceleration: Choices, Constraints and Optimal Strategies of Micro-Batch

Riqian Hu

HDU-ITMO Joint Institute, Hangzhou Dianzi University, Hangzhou, 310000, China
20321114@hdu.edu.cn

Abstract. As the scale of deep learning models continues to grow, large-scale models in machine vision and natural language processing (NLP) have achieved tremendous success. For instance, the current NLP giant GPT-3 has pushed the parameter count to the scale of billions. However, due to the significant surpassing of GPU physical memory limitations by large-scale deep neural networks, current strategies like data parallelism are no longer sufficient for model training. The latest pipeline parallelism strategies, such as the static layer partitioning of GPipe and PipeDream, as well as the dynamic layer partitioning of VPipe, have enabled model training segmentation and acceleration. In the current pipeline strategies like GPipe, the batch-splitting pipelining algorithm splits mini-batches on the same accelerator into overlapping computation stages, creating micro-batches to achieve pipelining. Users usually need to manually fine-tune the granularity of pipeline segmentation, i.e., micro-batch size (M), to determine the optimal value by observing changes in throughput. This article observes that M is not the smallest factor that affects throughput and proposes that batch size/micro-batch size (B/M) is the decisive factor that determines the changes in throughput. This article focuses on proving the rationality of B/M and quantitatively giving the selection range of B/M. For any given multi-GPU training scenario, by analyzing the optimal value of B/M in advance, the debugging cost can be reduced, and the throughput can be maximized quickly before training, thus accelerating the efficiency of multi-GPU parallelism.

Keywords: Multi-card training, Pipeline parallelism, GPipe, Micro-batch

1 Introduction

1.1 A Subsection Sample

Humans have never ceased to imagine and pursue artificial intelligence. How to construct a model that possesses human-level intelligence has always been a problem that AI scientists strive to solve [1]. In recent years, as researchers delve deeper into the field, an increasing number of outstanding models have been created, gradually approaching human-level perception and reasoning capabilities.

© The Author(s) 2024
Y. Wang (ed.), *Proceedings of the 2024 2nd International Conference on Image, Algorithms and Artificial Intelligence (ICIAAI 2024)*, Advances in Computer Science Research 115,
https://doi.org/10.2991/978-94-6463-540-9_101

In the field of computer vision, in 2012, AlexNet [2], based on convolutional neural networks, made remarkable achievements in image recognition, making Convolutional Neural Networks (CNN) the mainstream in machine recognition. In 2015, ResNet [3], designed to address the "degradation problem," successfully broke the limitation of network depth, marking a turning point where machine recognition accuracy surpassed human performance. In 2018, AmoebaNet [4], inspired by the principle of survival of the fittest in biological systems, was developed, paving the way for automated network design as a new approach. In 2020, Vision Transformer [5], based on a pure attention mechanism, was introduced into the field of machine recognition. Among them, Scaling ViT with 2 billion parameters pushed the recognition accuracy of ImageNet to new heights [6]. These models approach problem-solving from different angles and exhibit diverse architectures, but they converge on one aspect: the larger and deeper the model, the stronger its representational capacity and robustness [7].

The development of computer processing power and advancements in the field of deep learning are closely intertwined. To train larger and deeper models, there is a growing demand for increased computational power in computer hardware. In 2007, NVIDIA introduced CUDA (Compute Unified Device Architecture). Although CUDA was not originally designed specifically for deep learning, it provides highly optimized GPU parallel computing capabilities. This is particularly beneficial for model training that involves extensive matrix calculations and tensor operations. Parallel execution on GPUs is considered a viable solution for significantly accelerating computation speed in such scenarios. In 2009, Andrew Ng's team was among the first to leverage GPUs for model training, significantly accelerating the training speed [8]. Since then, the training of deep learning models has become deeply intertwined with the computational power provided by GPUs.

As the size of models continues to increase, a single GPU is no longer sufficient to meet the demands of training large-scale models. For instance, the massive GPT-3 model [9], which consists of 175 billion parameters, requires distributed training across multiple GPUs over the course of several weeks. However, relying solely on splitting the network layers for achieving model parallelism does not yield satisfactory results, as the problem of imbalanced activations within the layers significantly leads to performance degradation.

In this context, pipeline parallelism techniques have been introduced into model training and have been proven to be effective and reliable [10]. Common pipeline parallelism techniques include PipeDream [11] and GPipe [10]. PipeDream stores all activation tensors in the GPU memory and waits for the backward pass [11], while GPipe only retains the activation tensors involved in the communication part of the forward pass [10]. Both approaches belong to static partitioning. The latest VPipe technology can dynamically allocate the computational workload among GPUs by introducing a virtual layer into the existing pipeline parallelism pipeline and hardware [12]. This helps to avoid throughput bottlenecks caused by an imbalanced workload on individual GPUs, whether it is too heavy or too light.

Using GPipe, it is possible to represent a model as a sequence of layers, and groups of consecutive layers can be divided into cells. These cells are then assigned to separate accelerators. Building upon this partitioned arrangement, a new algorithm for pipeline

parallelism with batch splitting was proposed. The algorithm involves initially splitting a mini batch of training examples into smaller micro-batches, and then executing each set of micro-batches in a pipeline fashion across the cells. Synchronous mini-batch gradient descent is applied for training, wherein gradients are accumulated across all micro-batches within a mini-batch and applied at the end of each mini-batch [10].

Under the GPipe pipeline parallelism strategy, users need to define the size of the micro-batch (M), which determines the number of chunks a mini batch is divided into for pipeline parallelism [10]. Huang et al. chose M=1, 4, 32 in their GPipe experiment and stated that "Batch size (B) was adjusted to fit memory when $M>>K$ (number of partitions on accelerators)" but it is unknown what batch size values they chose. In the experimental process, it can be observed that M is not the only value that affects GPU throughput, and after investigation, batch size/micro-batch size (B/M) is an important factor that affects throughput. This article will demonstrate the important role of B/M and discuss methods for selecting the optimal B/M. By calculating the optimal B/M value, I can increase the training throughput of GPipe by 3% through adjusting hyperparameters.

2 Main Idea of GPipe

2.1 Concept of multi-card training

In deep learning, batch size refers to the number of samples used for training the model in a single iteration. Compared to using the entire dataset for one iteration, using a random subset of the training set for multiple iterations can better mitigate model overfitting. Both excessively large and small batch sizes have limitations on model training. An excessively large batch size may lead to out-of-memory (OOM) errors and result in overfitting. On the other hand, a very small batch size can result in poor generalization, leading to lower accuracy and other issues. Finding an appropriate batch size is crucial for balancing memory constraints, model performance, and generalization capabilities.

As the model size increases, I may encounter limitations in GPU memory that prevent us from training the entire model on a single GPU. Therefore, I adopt a strategy of partitioning the model, where a batch is divided into multiple mini batches, which is called model parallelism. Each mini batch is then assigned to a separate GPU for training, and the gradient information is passed from one GPU to the next. This partitioning strategy allows us to distribute the workload across multiple GPUs and overcome the memory limitations, enabling efficient training of large models. By exchanging gradient information between GPUs, the model parameters can be collectively updated during the training process.

2.2 Definition of micro-batch

In the context of deep neural networks, it is possible to represent any network as a sequence of L layers. By specifying the number of partitions, K, the sequence of L layers can be divided into K composite layers or cells [10], each of them called a mini batch. Fig. 1 below shows the procedure of L layers partition.

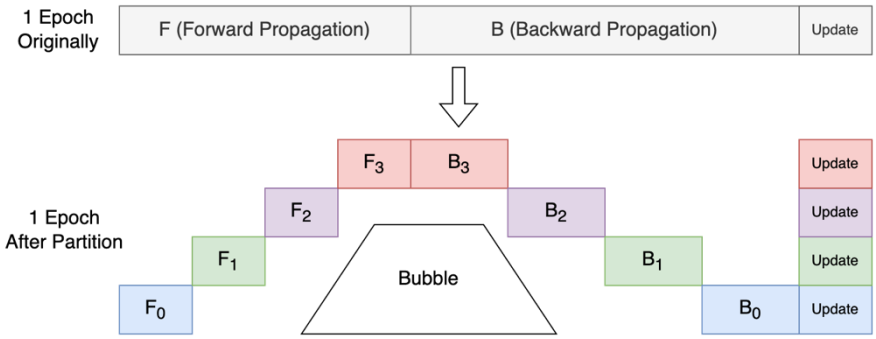

Fig. 1. Diagram of naive model parallelism strategy

As depicted in the above diagram, with the utilization of model parallelism, the massive number of parameters that would have exceeded the memory limit of a single GPU are divided and allocated across multiple GPUs, enabling parallel training of the model. However, the drawback of this model parallelism strategy is evident. As seen in each stage of GPU training, the GPU enters a stagnant state and waits until it receives the backward-propagated parameters. Consequently, the model parallelism strategy inevitably results in significant bubbles that lead to low efficiency.

GPipe introduced the strategy of pipeline parallelism to address the issue of low efficiency in model parallelism. GPipe states that if the K^{th} partition of the model contains layers i to j, it is possible to further divide the mini batch into micro-batches based on the following equation (Equation 1, 2), allowing different accelerators to process different micro-batches simultaneously.

$$W_K = w_i \cup w_{i+1} \cup \cdots \cup w_j \tag{1}$$

$$F_K = f_i \circ f_{i+1} \circ \cdots \circ f_j \tag{2}$$

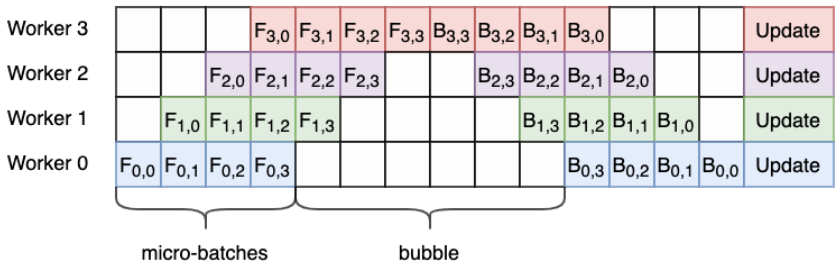

Fig. 2. Pipeline parallelism with divided micro-batches

With the theoretical foundation of partitioning, I can further divide each partition by splitting the mini batches assigned to each GPU into smaller micro-batches. These micro-batches serve as the fundamental units for computation. At a finer granularity, I can introduce a certain level of overlap in the GPU's computation cycles, thereby achieving

parallel computing effects. As shown in Fig. 2, this approach significantly reduces the efficiency loss caused by bubbles.

2.3 GPipe limitation

While GPipe is designed to minimize bubbles, it is inevitable to observe some degree of bubbles in the training process. To enhance training efficiency, it is important to minimize the bubble value. Although the exact duration of bubbles cannot be quantified, I can define a bubble ratio as the ratio of bubble time to the total training time of an epoch. This is represented by Equation 3, and a sketch map is provided in Fig. 3. By reducing the bubble ratio, I can improve the throughput of training. Note that I will use the abbreviation *BR* to refer to the bubble ratio.

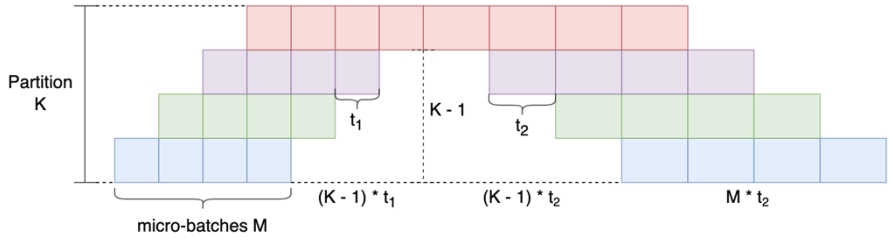

Fig. 3. Sketch of Equation 3

$$
\begin{aligned}
BR &= \frac{(K-1)t_1 + (K-1)t_2}{Mt_1 + (K-1)t_1 + (K-1)t_2 + Mt_2} \\
&= \frac{(K-1)(t_1 + t_2)}{(M+K-1)(t_1 + t_2)} \\
&= \frac{K-1}{M+K-1}
\end{aligned} \tag{3}
$$

As shown in the above equation, under the assumption of equal partitioning (as assumed in the GPipe paper), the value of *BR* is a constant that is independent of the forward or backward propagation time. It is solely determined by the values of *K* and *M*.

The GPipe paper suggests that when $M \geq 4 \times K$, the performance loss caused by bubbles can be neglected. From Equation 3, it can be deduced that under the condition $M \geq 4 \times K$, the *BR* will always remain below 20%. Theoretically, reducing the *BR* by approximately 80% represents a significant improvement in performance.

I conducted tests using the open-source *torchgpipe* library [13] with the AmoebaNet-D model on the ImageNet dataset. However, after testing several cases, I found that the experimental results differed from what was described in the GPipe paper. The optimal solution of $M \geq 4 \times K$ did not hold for all batch sizes. When testing with smaller batch sizes, whether using two, four, or eight GPUs (with *K* equal to the number of GPUs), the highest throughput was generally achieved at a relatively small value of *M*. The

training results are shown in Fig. 4, where empty values indicate that training was not possible due to insufficient GPU memory.

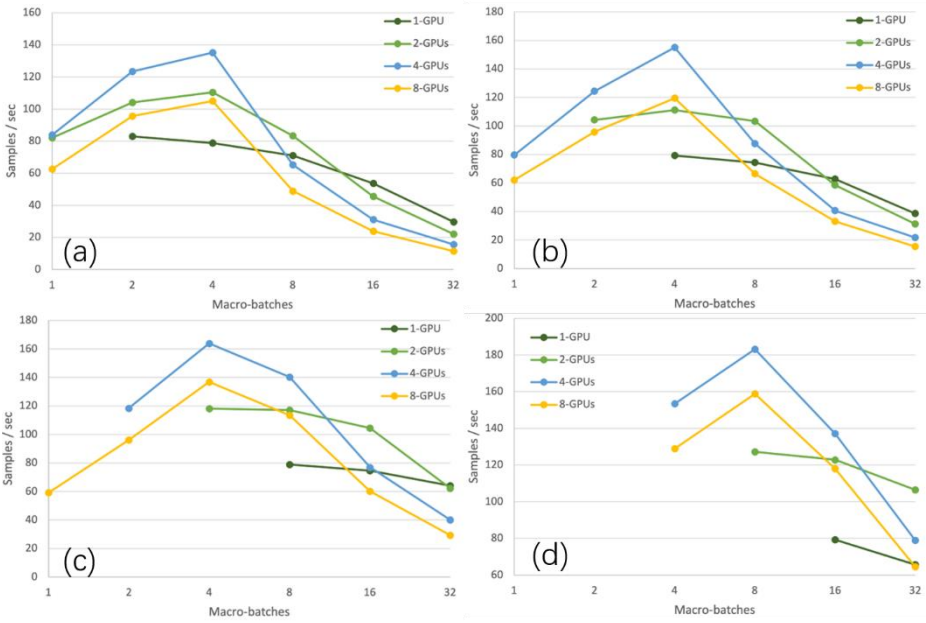

Fig. 4. Results of training throughput using GPipe
(a) B=96 (b) B=128 (c) B=256 (d) B=512

Upon observing the graph, several points can be noted. Firstly, as *B* increases, the peak throughput shifts towards larger values of *M* (towards the right). Additionally, it is evident that the value of *M* cannot be too large or too small. This indicates that although bubbles are reduced with sufficiently large *M* values, other performance losses are increased. The specific reasons behind these phenomena will be analyzed in the next section.

3 Find the optimized micro-batch

3.1 Reasons for the constraint on micro-batch size

There are several key factors that influence the selection of the *M*, which determines that the optimal value of *M* falls within a certain range and should not be too large or too small.

The reason for not choosing an excessively small value for *M* is quite evident: when *M* is too small, the main issue is that it leads to excessive GPU memory usage within one epoch, resulting in OOM errors. Moreover, the value of *M* is closely related to the *BR*. A larger value of *M* allows for a reduction in the *BR*, thereby improving GPU throughput.

The GPipe paper encourages us to increase the value of *M* as much as possible. However, based on the recent experimental results, it can be concluded that GPU throughput is sensitive to larger *M* values, as it rapidly decreases beyond a certain threshold. To verify the hypothesis of a "threshold," I continued testing the relationship between throughput and *M*. In Fig. 4, the granularity of *M* partitioning is relatively large (using powers of 2 as inputs). To further investigate, I conducted tests at a finer granularity (incrementing by 1 each time) for *M*, as shown in Fig. 5.

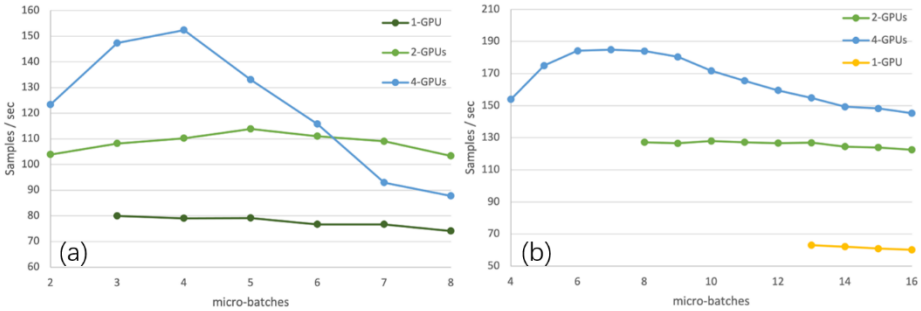

Fig. 5. Results of training throughput in specific samples
(a) B=128 (b) B=512

As depicted in the above graph, the optimal value of *M* is not a specific threshold but lies within a small range. Our goal moving forward is clear: I need to identify the optimal range for *M* that maximizes the training throughput of the model.

There are several reasons why *M* cannot be set to a very large value. After gaining a deep understanding of the GPipe principles, several factors are identified that limit the choice of *M* and lead to a decrease in throughput as *M* increases. The first reason is that GPipe introduces the method of re-materialization to reduce memory usage. It has been observed that recomputing certain values is more efficient than randomly reading them from memory. During the forward computation process, each accelerator exclusively retains the output activations at the partition boundaries, optimizing memory utilization [14]. The second reason is that as the value of *M* increases, there are more accumulations of gradients during each mini batch, leading to a performance cost. At the conclusion of every mini batch, the gradients obtained from all *M* micro-batches are collected and utilized to update the model parameters across all accelerators [10]. The third reason is that if batch normalization [15-20] is used in GPipe, the sufficient statistics of inputs during training are computed over each micro-batch and over replicas if necessary. Clearly, an increase in the value of *M* results in a higher number of batch normalization computations. The increased computations can diminish the advantages of GPU parallelism, leading to a performance cost.

3.2 Factors influencing throughput

Since GPipe assumes balanced partitions [10], I only need to set all balances to the same value. In this paper, our focus is on throughput rather than model training accuracy. Therefore, I can set the optimizer to a fixed value, using *AdamW* uniformly, and select three epochs. Finally, I calculate the average throughput.

To facilitate quick modification of variables, I have provided an interface in the command line, as shown in the code snippet below.

After conducting several experiments, I have plotted the relationship among *B*, *M*, and the number of GPUs, as shown in Fig. 6. The absence of data points in certain areas is due to OOM errors, preventing the retrieval of results.

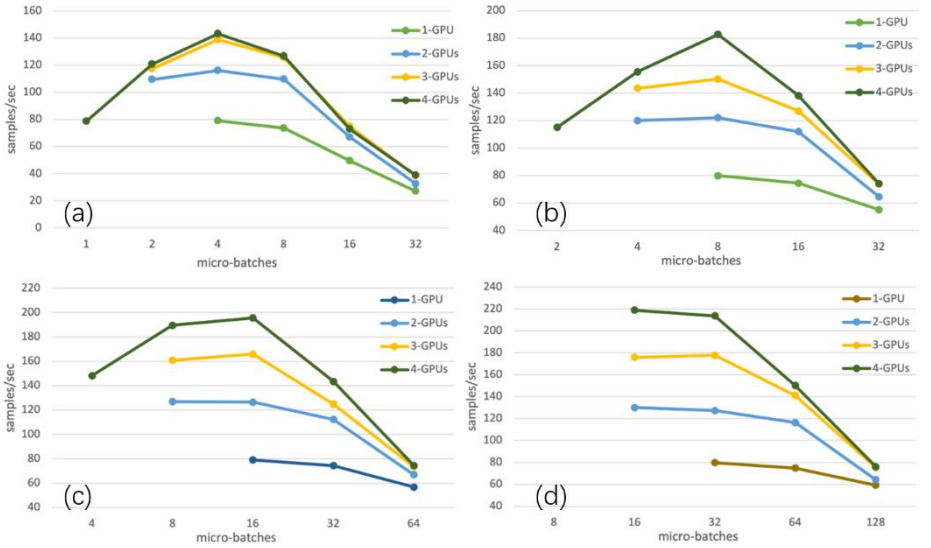

Fig. 6. Throughput with micro-batch sizes and GPUs
(a) B=128 (b) B=256 (c) B=512 (d) B=1024

From the four graphs above, several key observations can be made. First, as the batch size increases, the maximum throughput value consistently shifts towards the right. Although both the *B* and *M* can impact the throughput, the ratio *B/M* provides useful information. The maximum throughput value is consistently found near *B/M* = 32. Second, as the number of GPUs increases, the model training can utilize smaller micro-batches. With a smaller micro-batch range, the search space for the optimal throughput value can be expanded.

Meanwhile, as the batch size increases, the advantages of multi-GPU parallel computation become more evident. As shown in Fig. 7, when I set *B/M* to 32, I can observe a noticeable differentiation in throughput with increasing batch size in multi-GPU setups. When the batch size reaches 1024, the throughput of a single GPU remains relatively stable, while each additional GPU contributes to a 20-50% increase in throughput.

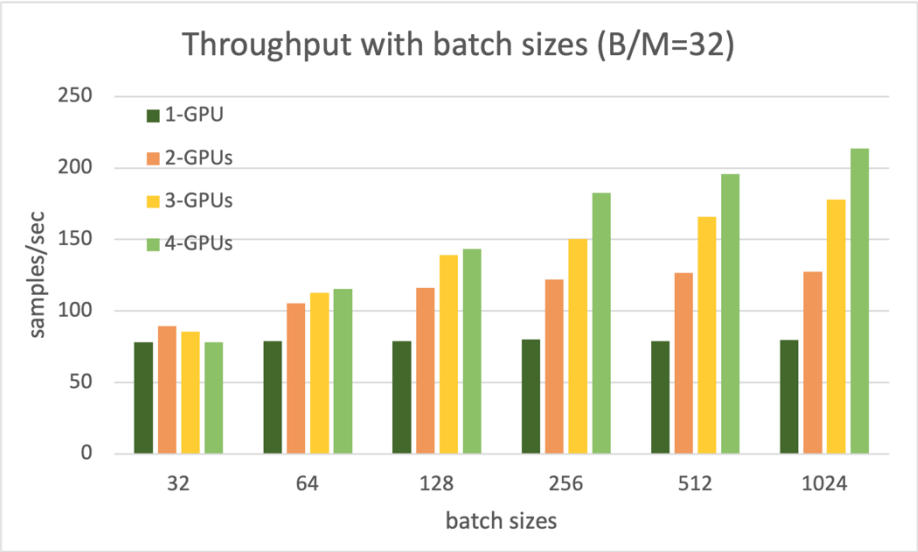

Fig. 7. Throughput with batch sizes under B/M=32

Therefore, the key to increasing GPU throughput is to find an optimal *B/M* value and appropriately increase the batch size while ensuring model training accuracy. In the next section, I will discuss how to find the optimal *B/M* value.

3.3 Find an optimized B/M

In deep learning, the determination of *M* typically depends on the GPU memory capacity and the size of the model. In general, to maximize training speed and efficiency, it is desirable to keep each GPU as busy as possible. This means using the largest feasible micro-batch size to leverage the parallel computing capabilities of the GPU without causing memory overflow.

GPU memory not only needs to store the model parameters but also the activations and gradients. An excessively large micro-batch size inputs many samples to the GPU at once, resulting in high memory usage and potential OOM errors. On the other hand, a very small micro-batch size can lead to underutilization of the Arithmetic Logic Units (ALUs), reducing the training efficiency of the model, as shown in Fig. 8.

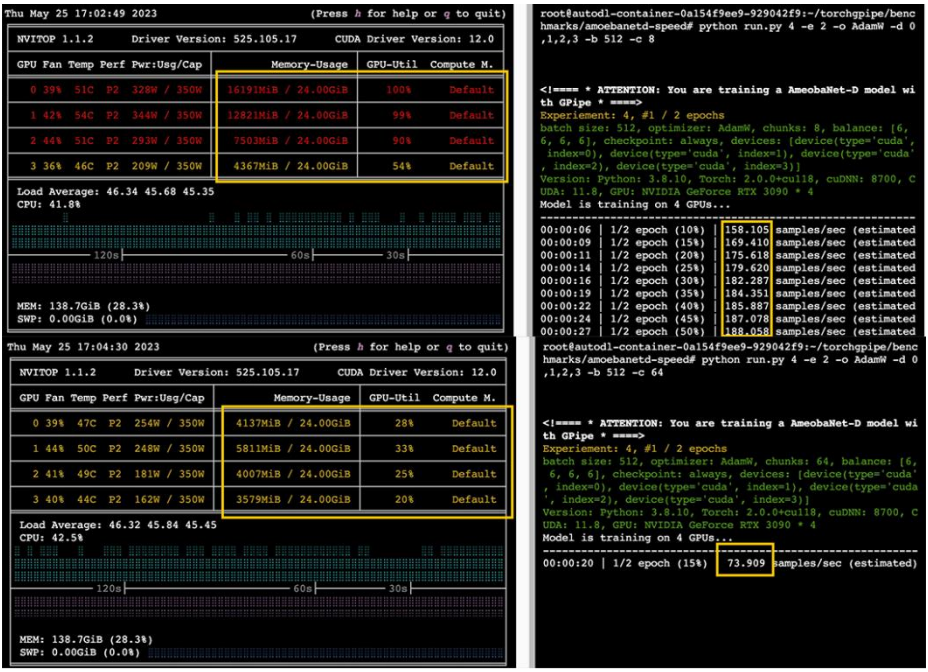

Fig. 8. Comparison of throughput on different memory usages

To select an appropriate *B/M*, it should be understanded what *B/M* represents for GPU. Firstly, to prevent overflow of memory on the video card, it is needed to manually specify an upper bound on the *B/M*. As GPipe supports re-materialization, assuming a batch size of B samples is evenly distributed over *M* micro-batches, the number of samples the GPU needs to process at once is *B/M*. If the size of a single sample is S, then *(B×S)/M*. Meanwhile, the GPipe paper also mentions that during the backward pass, the K^{th} accelerator recomputes the composite forward function F_K. Assuming the model has *L* layers, is divided into *K* partitions, and each partition has *N* mini-batches, with each mini-batch divided into *M* micro-batches. For recompute to proceed smoothly, the GPU must reserve at least *O(N+(N×L)/(M×K))* space to complete the recompute. Therefore, when selecting *M*, the rule should be followed which is shown in Inequation 4:

$$Max\left\{\frac{\mathrm{B}\times S}{M}, N+\frac{L}{K}\times\frac{N}{M}\right\} < GPU's\ Memory \tag{4}$$

This is just an estimate, and actual experimentation and tuning may be necessary, as memory usage may not be perfectly linear and may be affected by factors such as CUDA memory management. After determining an upper bound for *B/M*, it ensures that memory overflow will not occur.

From the recent testing, the optimal *B/M* for multi-GPU parallelism lies near the upper bound of *B/M* for a single GPU. This is because when near the upper bound for a single GPU, the memory utilization is highest, leveraging the advantages of GPU

parallel computing to the fullest. However, why could the optimal *B/M* for multi-GPU parallelism also lie to the left of the upper bound? This is because as *K* increases, the memory required for recomputing decreases, and the GPU can store more samples, causing the minimum value of *B/M* to shift to the left.

When searching for *B/M*, the following method can be used: first, set *M*=1, which is equivalent to not dividing into micro-batches. At this point, except for *B*, all variables in Equation 4 are constants. Then, the maximum theoretical value of *B* can be calculated, and thus *B/M* can be obtained. Although the theoretical value has been determined, the value of *B/M* still needs to be further determined, as the mechanism of CUDA memory allocation is far more complicated, and *B/M* can only be greater than or equal to the theoretical value.

As multiple GPUs are available, test benches with different *B* values on each of the GPUs can be synchronized, or stress testing can be used, and the GPU memory usage can be monitored in real-time through methods such as `nvidia-smi` or `torch.memory_allocated()`. By eliminating *B* values that can cause memory overflow, the maximum value of *M* can be obtained, which is now the actual maximum M value needed and is the desired *B/M*. A significant improvement in training throughput can be observed by scaling *B* and *M* proportionally. The *B/M* value can be fine-tuned and the optimal values of *B* and *M* can be chosen while ensuring training accuracy.

Table 1. Hardware and environment

Hardware / Software	Parameters and Version
CPU	Intel® Xeon® Gold 6330 2.00GHz
GPU	NVIDIA RTX 3090(24GB) × 8
Code Language	Python 3.8
ML Framework	PyTorch 2.0.0
Operation System	Ubuntu 20.04 LTS
CUDA	11.8

4 Testbench evaluation

In this chapter, to prove my point, I will conduct tests on a new platform. Table 1 below shows my test environment. To demonstrate that the optimal *B/M* can bring maximum throughput and is a fixed value, the following experiments are conducted. 1. Fix *M* and adjust *B* to observe changes in throughput. 2. Fix *B* and adjust *M* to observe changes in throughput. It was found that although *B* and *M* are two independent quantities, the size of throughput is always related to the ratio of *B/M* and will always achieve the maximum throughput at the same *B/M* value.

4.1 Check the optimal B/M value

An experiment was initially conducted by fixing *M* at 32 (as demonstrated in the Huang et al. paper) and adjusting *B*. As demonstrated in graph (a) of Fig. 9, it was observed that the maximum throughput was achieved when the *B/M* ratio fell within the range of 40-50. Further tests are conducted within this range, as presented in graph (b) of Fig. 9, revealing that the optimal *B/M* ratio was approximately 45.

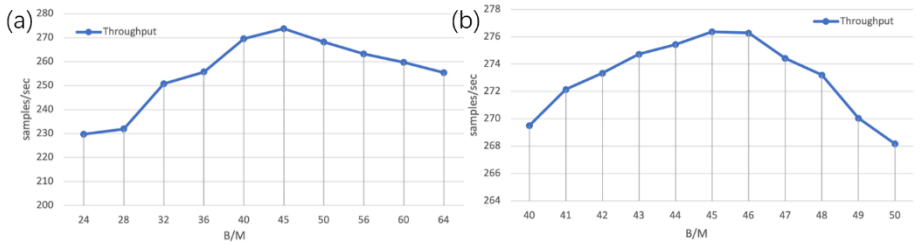

Fig. 9. Throughput variance with B/M based on M=32

Subsequently, *B* was fixed at 1600 and *M* was adjusted. As presented in graph (a) of Fig. 10, it was observed that the maximum throughput was achieved when the *B/M* ratio fell within the range of 44-52. Further tests are conducted within this range, as demonstrated in graph (b) of Fig. 10, revealing that the optimal *B/M* ratio was approximately 46.

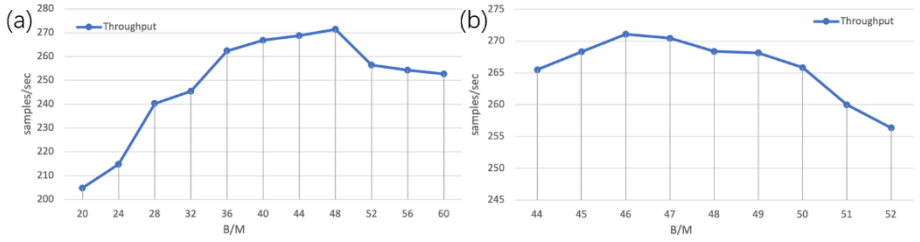

Fig. 10. Throughput variance with B/M based on B=1600

4.2 Check on broader batch sizes

To demonstrate that the same *B/M* is effective for all batch sizes, I conducted the following test, as shown in Fig. 11. The results showed that the *B/M* value that achieved the maximum throughput is a fixed value (strictly speaking, a small range of values), and if *B* does not exceed the memory limit, the value of *M* can be determined by *B*. With such a constraint relationship, the number of debugging attempts can be reduced and the maximum throughput of training can be quickly determined.

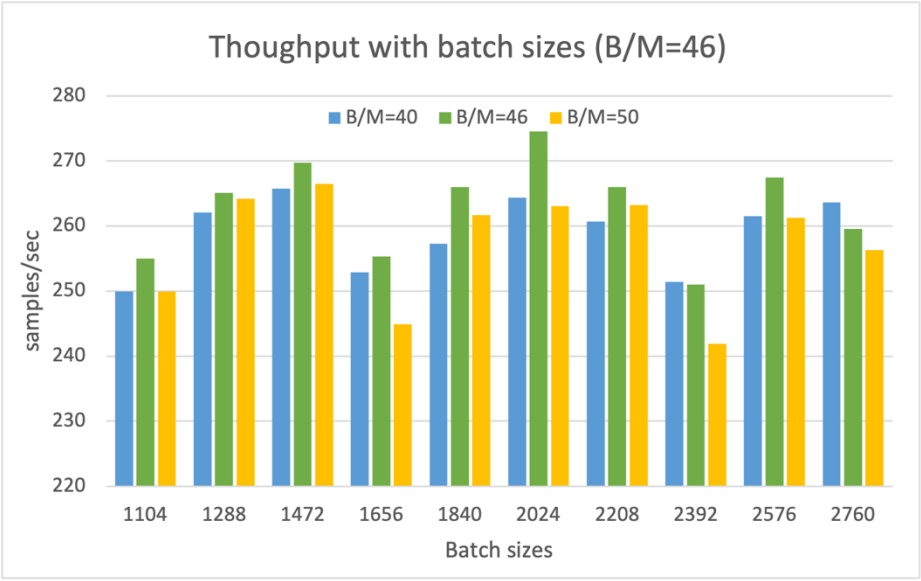

Fig. 11. Test bench on broader variants B and M

Of course, determining the optimal *B/M* is a process that can be influenced by various factors such as different GPUs, CUDA versions, and PyTorch versions that affect memory allocation to varying degrees. Only through testing can the ideal *B/M* value for training a specific model be determined.

4.3 Compared to the baseline

On this basis, I conducted additional experiments with eight cards to compare with the baseline. With the *B/M* value that achieved the maximum throughput, the experimental data are compared with the results presented in the Huang et al. paper. The results are shown in Table 2.

As shown in Table 2, the experimental variable is the number of GPUs (partition *K*), and the tested model is AmoebaNet-D. In comparison to the results presented in the paper by Huang et al., the experimental results of GPU throughput obtained purely through *B/M* tuning, without employing additional training techniques or utilizing new algorithms, are as follows.

Table 2. Comparison of optimized B/M and results in GPipe on AmoebaNet-D

K	Baseline (Huang et al.)	Optimized B/M	Improvement
1	/	0.67×	/
2	1×	1.03×	3.00%
4	1.7×	1.75×	2.94%
8	2.7×	2.79×	3.33%

After testing, by applying the optimal *B/M* value, the training throughput by 3% can be increased by adjusting the hyperparameters without changing the network structure and algorithm. Of course, improving GPipe is not the main content of this article.

5 Conclusion

Based on the facts described in the GPipe paper, this article analyzes the role of the hyperparameter micro-batch (*M*) in GPipe pipeline parallelism and innovatively proposes that *M* is not the smallest factor affecting GPU throughput, but rather the ratio of batch size (*B*) to micro-batch size, i.e., *B/M*. In the section of finding the optimal B/M, this paper explains how to select the optimal B/M technique, and quantitatively provides inequalities to define the range of values for B and M. Finally, this paper also demonstrates that the impact of B/M on GPU training throughput is global. That is, by constraining the optimal B/M ratio, it is possible to reduce the number of micro-batches debugging attempts and maximize GPU training throughput within a limited number of attempts, thereby increasing the training speed. However, this paper still has potential limitations. The optimal throughput does not necessarily mean that the accuracy of model training can be optimized, and future research can discuss the relationship between the two.

References

1. Minsky, M.: Steps toward artificial intelligence. Proceedings of the IRE. 49, 8–30 (1961).
2. Krizhevsky, A., Sutskever, I., Hinton, G.E.: ImageNet classification with deep convolutional Neural Networks. Communications of the ACM. 60, 84–90 (2017).
3. He, K., Zhang, X., Ren, S., Sun, J.: Deep residual learning for image recognition. 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR). (2016).
4. Real, E., Aggarwal, A., Huang, Y., Le, Q.V.: Regularized evolution for Image Classifier Architecture Search. Proceedings of the AAAI Conference on Artificial Intelligence. 33, 4780–4789 (2019).
5. Dosovitskiy, A., Beyer, L., Kolesnikov, A., et al.: An image is worth 16x16 words: Transformers for image recognition at scale, https://arxiv.org/abs/2010.11929.
6. Zhai, X., Kolesnikov, A., Houlsby, N., Beyer, L.: Scaling vision transformers. 2022 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR). (2022).
7. Kaplan, J., McCandlish, S., Henighan, T., et al.: Scaling laws for neural language models, https://arxiv.org/abs/2001.08361.
8. Raina, R., Madhavan, A., Ng, A.Y.: Large-scale deep unsupervised learning using graphics processors. Proceedings of the 26th Annual International Conference on Machine Learning. (2009).
9. Brown, T.B., Mann, B., Ryder, N., Subbiah, M., et al.: Language models are few-shot learners, https://arxiv.org/abs/2005.14165.
10. Huang, Y., Cheng, Y., Bapna, A., et al.: GPipe: Efficient training of giant neural networks using pipeline parallelism, https://arxiv.org/abs/1811.06965.
11. Narayanan, D., Harlap, A., Phanishayee, A., et al.: PipeDream: Generalized pipeline parallelism for DNN training. Proceedings of the 27th ACM Symposium on Operating Systems Principles. (2019).

12. Zhao, S., Li, F., Chen, X., Guan, X., et al.: VPipe: A virtualized acceleration system for achieving efficient and scalable pipeline parallel DNN training. IEEE Transactions on Parallel and Distributed Systems. 33, 489–506 (2022).
13. Kim, C., Lee, H., Jeong, M., et al.: Torchgpipe: On-the-fly pipeline parallelism for training giant models, https://arxiv.org/abs/2004.09910.
14. Chen, T., Xu, B., Zhang, C., Guestrin, C.: Training deep nets with sublinear memory cost, https://arxiv.org/abs/1604.06174.
15. Zhang, B., Zhou, Z., Cao, W., Qi, X., Xu, C., Wen, W.: A new few-shot learning method of bacterial colony counting based on the edge computing device. Biology. 11, 156 (2022).
16. Ioffe, S., Szegedy, C.: Batch normalization: Accelerating deep network training by reducing internal covariate shift, https://arxiv.org/abs/1502.03167.
17. Lyu, Y., Yang, Z., Liang, H., Zhang, B., Ge, M., Liu, R., Zhang, Z., Yang, H.: Artificial Intelligence-assisted Fatigue Fracture Recognition based on morphing and fully convolutional networks. Fatigue & Fracture of Engineering Materials & Structures. 45, 1690–1702 (2022).
18. Wang, L., Yang, Y., Min, R., Chakradhar, S.: Accelerating deep neural network training with inconsistent stochastic gradient descent. Neural Networks. 93, 219–229 (2017).
19. Ambrogio, S., Narayanan, P., Tsai, H., Shelby, R.M., Boybat, I., di Nolfo, C., Sidler, S., Giordano, M., Bodini, M., Farinha, N.C., Killeen, B., Cheng, C., Jaoudi, Y., Burr, G.W.: Equivalent-accuracy accelerated neural-network training using analogue memory. Nature. 558, 60–67 (2018).
20. Mahmoud, M., Edo, I., Zadeh, A.H., Mohamed Awad, O., Pekhimenko, G., Albericio, J., Moshovos, A.: TensorDash: Exploiting sparsity to accelerate deep neural network training. 2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO). (2020).

Open Access This chapter is licensed under the terms of the Creative Commons Attribution-NonCommercial 4.0 International License (http://creativecommons.org/licenses/by-nc/4.0/), which permits any noncommercial use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.