Performance Analysis of Parallel Computing in Image Classification

Yicong Hao*

Department of Software Engineering, McGill University, Montreal, H3A 0G4, Canada

*Corresponding Author Email: yicong.hao@mail.mcgill.ca

Abstract. The use of parallel computing can speed up the training of deep learning models. The traditional neural network model ResNet is chosen for testing in this paper on the effectiveness of data parallelism in image classification, and test data is provided in a 6-GPU environment. In this paper, it is suggested that various factors should be considered when building affordable hardware configurations to expedite model training in practical application scenarios. The communication costs are not insignificant because today's large computing clusters are primarily offered via cloud computing. Another crucial point to remember is that the number of CUDA cores is the primary hardware foundation for GPU acceleration technology. Therefore, acceleration may not be affected by more incredible video memory or fewer CUDA cores for some particular graphics cards. In addition, the issue of beyond-model performance in parallel computing is another issue that must be disregarded. Due to the parallel strategy's limitations, it is essential to tweak the super parameters while speeding up model training. The model's performance is more likely to be ensured by the lower learning rate and Batch size. This paper's experimental conclusion can support building an appropriate hardware configuration scheme.

Keywords: component; parallel computing, data parallel, deep learning; distributed data parallel, image classification.

1. Introduction

Deep learning is a specified field of machine learning, ML, and artificial intelligence, AI, that obtains some knowledge by imitating the way humans do it [1]. In deep learning, the crucial concepts are gradient and gradient descent. In mathematical terms, the gradient is defined as the derivative of some function that has more than one input variable [2]. In the field of deep learning and other subsets of machine learning, gradient evaluates the change in weights of the function when a change occurs for the input [3]. It can be viewed as a function's slope since they have the same behavior [4]. The lower the gradient, the gentler the slope, and the slower learning of the model [5]. However, if the slope is 0, no learning occurs for the model [6]. Furthermore, gradient descent is utilized on differentiable functions as an optimization algorithm to find their local minimums. One of the most popular and common gradient descent algorithms used in machine learning and deep learning is stochastic gradient descent (SGD) [7]. SGD picks data points randomly from the dataset for each iteration to simplify the computation [8]. However, the simplification comes with a cost, it takes longer to converge, and the gradients represented are less accurate. This can be solved by creating larger batch sizes or using an entire dataset.

The gradient descent method is a general solution for deep learning. However, due to the scale of the deep learning model, it is usually necessary to search for the optimal local solution in an ultra-high dimensional space. As a result, the deep learning model requires extremely high computing resources. To maximize the performance of deep learning algorithms, researchers parallelize them through data and model parallelism. Data parallelism is used to solve the problem of not being able to fit an extensive dataset into the memory. The datasets are split into N parts on N different GPUs, then assigned to parallel computational machines. In data parallelism, the models can be either synchronously or asynchronously trained. The model sends different data parts into each respective accelerator as part of synchronous training. A complete copy of the model is individually trained on the given part of the data. All the parts simultaneously start forward and calculate their respective gradient and output. To gather all trainable parameters from the processing units, the synchronous

training utilizes an all-reduce algorithm [9]. While synchronous training is superior in various aspects, it is also relatively harder to scale. On the other hand, asynchronous processors do not care about the progress of the others, therefore, eliminating the need to wait for each other. This is especially advantageous when working with more limited devices. For example, devices that are comparatively smaller and more limited. While synchronous training is more suited if the devices are more vital and with a powerful connection. In this paper, the performance of data parallelism in image classification is investigated. The data are trained in a 6 GPU environment with the classical neural network model ResNet. We will investigate the possible factors that can contribute increase model training efficiency when working with real world data and hardware. In addition, in the modern computer science industry, most large datasets are processed using cloud computing, and the communication cost among the computing clusters cannot be overlooked. Furthermore, the hardware aspect of the GPU acceleration technology is also crucial. It is important to acquire and apply the correct number of CUDA cores. Thus, it is needed to study the correlation between the significant video memory and the number of CUDA cores on the graphic cards used. Last but not least, knowing that parallel computing cannot be the only factor influencing the model performance of the algorithm. We will also try to alter the super parameters, namely the learning rate and the batch size, to see their effect on accelerating the training speed of the model.

2. Methods

2.1. Data Parallel

Data parallel is comparatively easy to use because it is a multithread and single process, only functional on a single machine [9]. Although it can still enable multiple GPUs, data parallel is not capable of scaling more than one machine [10]. Data parallel requires us to divide the dataset into different chunks and then send them to different processing units to full fill the computation. When dividing the dataset into different batches, the PyTorch nn.DataParallel algorithm would need the number of GPUs to be an aliquot of the batch size [11]. It will place the network model parameters on GPU0 by default, and therefore copy the parameters to other GPUs and train them at the same time. This can be problematic, for reasons of uneven work distribution for different graphic cards and memory. In other words, GPU0 will be utilizing much more times of memory compared to the other CPUs. Thus, theoretically slower than distributed data parallel. Furthermore, the communication cost among the multiple threads, along with the overhead also increases the runtime for data parallel [12]. Considering that all the data communication is encapsulated in the same one process, the time taken to run the algorithm using data parallel may do worse than the sequential implementation.

2.2. Distributed Data Parallel

The distributed data parallel approach ensures all the processing units have the same weight, equal work, and share working progress for a set duration. Therefore, the cost of communication is lowered. The distributed data parallel system solves the data uneven distribution problem by isolating the training processes for each process and only exchanging a limited amount of information periodically. The primary GPU which is responsible for synchronizing, making copies, model loading, and writing logs of the processes, is called the master node. The master node copies the model to all the GPUs. When all the process will finish their calculation, the process with rank=0 gather and average them out, and broadcast the results to all processes [13]. Distributed data parallelism is the multi process that can be used on one or multiple machines [14].

This paper decides to use the distributed data parallel method to perform our experiments. Since between nn.DataParallel and nn.DistrbutedDataParallel, the DDP container, is officially recommended by the developers of PyTorch in the official documents because it is more flexible and faster [15]. The implementation of the distributed data parallelism can be concluded in five core steps. First of all, create up to six process groups, for we have six GPUs. Secondly, divide the dataset into up to six respective groups. Thirdly, wrap the deep learning model with distributed data parallel.

Fourth, run the deep learning algorithm, train and test the model the same way as working with a sequential program, and record the result. Last but not least, clean up the six process groups created. Thanks to the developers of PyTorch, most of the processes can be done quickly with the functions ready to use from the PyTorch library.

2.3. ResNet

ResNet is the acronym for residual neural network, one of the most common CNN (Convolutional Neural Network) models, an artificial network that can go into hundreds of layers deep. It is built referencing the VGG19 network, inheriting its three by 3 convolutional layers design, and introducing residual units with the shortcut connection mechanism [16]. One of the most critical design cores was that when the feature map is increased to double its size, the number of feature maps will be halved, ensuring the network layers' complexity. A standard resNet model can be implemented with double or triple layers skips that contain batch normalization and nonlinearities, such as ReLu.

2.3.1 ResNet18

ResNet18 is an architecture with 72 layers and 18 deep layers [17]. The goal of ResNet18 is to enable the function of large-scale convolutional layers. Hence, makes it perfect for our experiment, and helps with the stress test of different parallel algorithm performances on multiple GPUs. It is supported by PyTorch, with parameters of weights, progress, and **kwargs [11]. The number 18 in the name indicates the depth of the network, meaning there are 18 layers of weighted layers, including the convolutional layers and connected layers, excluding the pooling layers and BN layers.

3. Experimental Results and Analysis

3.1. Data description

This paper uses the CIFAR-10 dataset to test the data parallel scheme. The data set consists of sixty thousand 32*32 coloured pictures, comprising ten categories, six thousand pictures in each category. There are fifty thousand training pictures and ten thousand testing pictures. The dataset is divided into six blocks in total. Among the six blocks, five of them are training blocks and the other one is the testing block. All of the blocks contain ten thousand pictures respectively. The test block contains one thousand pictures randomly selected from every category. The training blocks contain random residual images, but some may contain more images for one category than others. The training blocks contain five thousand images from each category. These classes are mutually exclusive, and pictures that appear in one class will not appear in any other class. The hyperparameters of the model are configured as, Epoch = 20, learning rate = 0.01.

3.2. Results and analysis

First and foremost, this paper tests the influence of the number of GPUs on the model's performance. The resulting time taken and accuracy reached are shown in Table 1, where N represents the number of GPUs used. The time taken was not reduced as hypothesized before. It did not reduce the time taken as soon as the number of graphic cards increased. Instead, the time increased then dropped. This makes sense because communication costs come into play when running on multiple cards, increasing the time taken. That is why, when N equals 2, the accuracy of the computation remained the same, while the time taken increased by 32.73 s. Furthermore, as it sees a shorter time taken for the computation, the accuracy was dropped simultaneously. This paper thought that with more graphic cards participating in the data processing, it would be natural to assume that the accuracy would increase. However, this was not the case. After extensive research, the problem has to do with the nature of deep learning. More specifically, the dependency of deep learning on gradient descent. As mentioned in the previous text, stochastic gradient descent finds a local minimum of some randomly picked points. When multiple GPUs are present, each one will pick their random data points

and make predictions. The process of averaging the local minimum becomes noisier, reducing accuracy.

Table 1. The influence of the number of GPUs on the model's performance

	N=1	N=2	N=4	N=6
Times （s）	99.22s	131.95s	81.84s	65.08s
Accuracy	72%	72%	68%	65%

When testing the algorithm's performance with only one GPU and varying the batch size, this paper gathers and organizes results as in Table 2. In general, the time taken for the trials with larger batch sizes was reduced compared to the default batch size of 64. In addition, the accuracy dropped as the batch size grew. Unlike the continuously lower accuracy, the run time reduction was not proportional to the increase in batch size. The shortest run time was observed when the batch size was 256, 40.22 seconds faster than the default batch size and 16s shorter than the 1024 batch size trial. This is expected because the larger batch sizes will take longer to go through and find a local minimum. On the other hand, the reduction in accuracy was unexpected. From the hardware's perspective, there might be problems with the cooperation between the video memory and the CUDA. Alternatively, the performance did not match the batch size. It can even result from using inappropriate models and parameters for the tests.

Table 2. The performance difference of the single card under different batch_size settings.

Batch_size	64	256	512	1024
Times	99.22s	59s	74.54s	75s
Acc	72%	61%	52%	45%

To further verify the impact of batch_size on the model performance. This paper tests the performance of the 6-card parallel model under different batch_size settings, as shown in Table 3. Despite the generally lower accuracy and shorter time when working with six GPUs, the behaviour was like that of the experiment conducted with only one graphic card. Larger batch sizes led to lower accuracy, and the run time is optimal when the batch size is 256.

Table 3. The performance of the 6-card parallel model under different batch_size settings

Batch_size	64	256	512	1024
Times	65.08s	32.05s	34.55s	37.48s
Acc	65%	52%	43%	32%

Based on the above experimental results, this paper argues that the data parallel strategy can considerably boost the training speed of the model without considering the communication cost. However, as the current large-scale computing clusters are usually provided in the form of cloud computing, the communication cost cannot be ignored. Then another critical issue to consider is that the main hardware foundation of GPU acceleration technology currently is the number of CUDA cores. Therefore, more significant video memory and a lower number of CUDA cores for some specific graphics cards may not affect acceleration. The above conclusions can be applied to build a more cost-effective hardware configuration scheme. In addition, another problem that cannot be ignored is the problem beyond the model's performance in parallel computing. Due to the limitation of parallel strategy, while speeding up the training speed of the model, the setting of super parameters is critical. The lower learning rate and Batch_size is more conducive to ensuring the model's performance. However, on the other hand, it will also affect the training speed.

4. Conclusion

This paper studies the performance of data parallelism in image classification, selects the classical neural network model ResNet for testing, and provides the test data in a 6-GPUs environment. In this paper, it is considered that many factors should be considered to construct cost-effective hardware configurations to speed up model training under real application conditions. Since today's large computing clusters are primarily provided in the form of cloud computing, the communication costs are not negligible. Another critical point to consider is that the main hardware foundation of GPU acceleration technology is the number of CUDA cores. Therefore, more significant video memory and a lower number of CUDA cores may not affect acceleration for some specific graphics cards. In addition, another problem that cannot be ignored is the problem beyond model performance in parallel computing. Due to the limitation of the parallel strategy, the adjustment of super parameters is critical while accelerating the model's training speed. The lower learning rate and Batch_size is more likely to help ensure the model's performance.

References

[1] Brownlee, J. (2021, October 11). What is a gradient in machine learning? Machine Learning Mastery. Retrieved September 24, 2022, from https://machinelearningmastery.com/gradient-in-machine-learning/#:~:text=Gradient%20is%20a%20commonly%20used,learning%20algorithms%20use%20gradient%20information.

[2] Chatterjee, P. (2022, April 25). Data parallelism vs. model parallelism - how do they differ in distributed training? Analytics India Magazine. Retrieved September 24, 2022, from https://analyticsindiamag.com/data-parallelism-vs-model-parallelism-how-do-they-differ-in-distributed-training/

[3] Distributeddataparallel. DistributedDataParallel - PyTorch 1.12 documentation. (n.d.). Retrieved September 24, 2022, from https://pytorch.org/docs/stable/generated/torch.nn.parallel.DistributedDataParallel.html

[4] Getting started with distributed data parallel. Getting Started with Distributed Data Parallel - PyTorch Tutorials 1.12.1+cu102 documentation. (n.d.). Retrieved September 24, 2022, from https://pytorch.org/tutorials/intermediate/ddp_tutorial.html#:~:text=Comparison%20between%20DataParallel%20and%20DistributedDataParallel&text=First%2C%20DataParallel%20is%20single%2Dprocess,%2D%20and%20multi%2D%20machine%20training

[5] Gradient descent: An introduction to 1 of Machine Learning's most popular algorithms. Built In. (n.d.). Retrieved September 24, 2022, from https://builtin.com/data-science/gradient-descent

[6] He, K., Zhang, X., Ren, S., & Sun, J. (2015, December 10). Deep residual learning for image recognition. arXiv.org. Retrieved September 24, 2022, from https://arxiv.org/abs/1512.03385

[7] Kathuria, A. (2020, December 18). Intro to optimization in Deep learning: Gradient descent. Paperspace Blog. Retrieved September 24, 2022, from https://blog.paperspace.com/intro-to-optimization-in-deep-learning-gradient-descent/

[8] Mao, L. (2019, May 23). Data parallelism VS model parallelism in distributed deep learning training. Lei Mao's Log Book. Retrieved September 24, 2022, from https://leimao.github.io/blog/Data-Parallelism-vs-Model-Paralelism/

[9] Mishra, A. (2019). Machine learning in the AWS cloud: Add intelligence to applications with Amazon Sagemaker and Amazon Rekognition. Amazon. Retrieved September 24, 2022, from https://docs.aws.amazon.com/sagemaker/latest/dg/model-parallel-intro.html

[10] Optional: Data parallelism. Optional: Data Parallelism - PyTorch Tutorials 1.12.1+cu102 documentation. (n.d.). Retrieved September 24, 2022, from https://pytorch.org/tutorials/beginner/blitz/data_parallel_tutorial.html

[11] Resnet18. resnet18 - Torchvision main documentation. (n.d.). Retrieved September 24, 2022, from https://pytorch.org/vision/main/models/generated/torchvision.models.resnet18.html

[12] Single-machine model parallel best practices. Single-Machine Model Parallel Best Practices - PyTorch Tutorials 1.12.1+cu102 documentation. (n.d.). Retrieved September 24, 2022, from https://pytorch.org/tutorials/intermediate/model_parallel_tutorial.html

[13] Srinivasan, A. V. (2019, September 7). Stochastic gradient descent - clearly explained!! Medium. Retrieved September 24, 2022, from https://towardsdatascience.com/stochastic-gradient-descent-clearly-explained-53d239905d31

[14] Deeply. (2020, April 22). Dataparallel vs distributeddataparallel. PyTorch Forums. Retrieved October 4, 2022, from https://discuss.pytorch.org/t/dataparallel-vs-distributeddataparallel/77891/3

[15] Adaloglou, N. (2022, April 14). How distributed training works in pytorch: Distributed data-parallel and mixed-precision training. AI Summer. Retrieved October 4, 2022, from https://theaisummer.com/distributed-training-pytorch/

[16] Namespace-Pt. (2021, November 29). A comprehensive tutorial to pytorch distributeddataparallel. Medium. Retrieved October 4, 2022, from https://medium.com/codex/a-comprehensive-tutorial-to-pytorch-distributeddataparallel-1f4b42bb1b51

[17] Residual network. Residual Network - an overview | ScienceDirect Topics. (n.d.). Retrieved October 4, 2022, from https://www.sciencedirect.com/topics/computer-science/residual-network