Wheel: Accelerating CNNs with Distributed GPUs via Hybrid Parallelism and Alternate Strategy

Xiaoyu Du[1], Jinhui Tang[2,*], Zechao Li[2], Zhiguang Qin[1]
[1]University of Electronic Science and Technology of China, Chengdu, Sichuan, China
[2]Nanjing University of Science and Technology, Nanjing, Jiangsu, China
duxiaoyu@cuit.edu.cn,jinhuitang@njust.edu.cn,zechao.li@njust.edu.cn,qinzg@uestc.edu.cn

ABSTRACT

Convolutional Neural Networks (CNNs) have been widely used and achieve amazing performance, typically at the cost of very expensive computation. Some methods accelerate the CNN training by distributed GPUs those deploying GPUs on multiple servers. Unfortunately, they need to transmit a large amount of data among servers, which leads to long data transmitting time and long GPU idle time. Towards this end, we propose a novel hybrid parallelism architecture named 'Wheel' to accelerate the CNN training by reducing the transmitted data and fully using GPUs simultaneously. Specifically, Wheel first partitions the layers of a CNN into two kinds of modules: convolutional module and fully-connected module, and deploys them following the proposed hybrid parallelism. In this way, Wheel transmits only a few parameters of CNNs among different servers, and transmits most of the parameters within the same server. The time to transmit data is significantly reduced. Second, to fully run each GPU and reduce the idle time, Wheel devises an alternate strategy deploying multiple workers on each GPU. Once one worker is suspended for receiving data, another one in the same GPU starts to execute the computing task. The workers in each GPU run concurrently and repeatedly like Wheels. Experiments are conducted to show the outperformance of the proposed scheme over the state-of-the-art parallel approaches.

CCS CONCEPTS

• **Computing methodologies** → **Parallel computing methodologies**; **Computer vision**; **Neural networks**;

KEYWORDS

CNN, Acceleration, Distributed GPUs, Hybrid Parallelism, Alternate Strategy

MM '17, October 23–27, 2017, Mountain View, CA, USA
© 2017 Association for Computing Machinery.
ACM ISBN 978-1-4503-4906-2/17/10...$15.00
https://doi.org/10.1145/3123266.3123435

1 INTRODUCTION

Recently, convolutional neural networks (CNNs) have been widely used and achieved extraordinary performance in computer vision and multimedia areas. Various state-of-the-art CNN models, such as Alexnet [17], GoogLeNet [26], VGG [24], ResNet [12], and transfer deep networks [27], have been proposed to further improve the performance. These CNN models usually contain tens of millions of parameters to be trained. Furthermore, to train an effective CNN model, usually massive training data and a large number of iterations (tens or hundreds of thousands) are required. That is, training CNN models is very computationally expensive. Real-world applications may suffer from the low training speed of these models. Consequently, it is of practical importance and urgency to accelerate the CNN training.

There have been various works to accelerate CNNs. Some methods study the problem of accelerating CNNs in term of algorithms, such as the low-rank approximation [29]. They are efficient to accelerate the computing on one or two layers and whole (but shallower) models. And then some other works were proposed to accelerate the CNN training by optimizing the implementations [5, 13, 18]. To get ultimate efficiency, the local program optimization is widely used in CNN tools such as Caffe [13]. Caffe implements matrix and vector computing based on Basic Linear Algebra Subprograms (BLAS), which is a set of routines providing common linear algebra operations to accelerate the matrix computing process without any extended devices [18]. The training of CNNs can be significantly accelerated by implementing computations on parallel computing units such as the NVIDIA Graphical Processing Unit (GPU).

Unfortunately, training deep CNN models on a single GPU is also too slow in practice. Many works are proposed to use multiple GPUs in parallel to further compare the efficiency [10, 31]. Thus many parallelization schemes using the multiple GPUs have been proposed. They are roughly categorized into data parallelism and model parallelism [9]. In model parallelism, CNNs are partitioned into several parts, and different parts are trained on different workers. These methods should be specially customized for different requirements, which limits their applications. Differently, in data parallelism, all the workers train CNN with different input data and synchronize their parameters with a parameter center. The parameter center always uses the asynchronous stochastic gradient descent (ASGD) algorithm [25] to update

*Corresponding Author

the parameters. Most current available parallel tools such as MXNet [2] and Torch7 [5] are based on the data parallelism scheme. Beside, the approaches proposed in [8, 16] use the hybrid parallelism scheme composed of model parallelism and data parallelism. But they actually use model parallelism since the data are synchronized finally and the structures are hard to extend. However, the aforementioned methods are mainly implemented on one server. The number of GPUs that can be deployed on one server is limited. Thus the improved speedup ratio is limited.

It is necessary to design the parallelization schemes by using distributed GPUs deployed on multiple servers. Several distributed GPU frameworks were designed as middlewares to integrate multiple GPUs among servers automatically [11, 14]. However, it is time-consuming to synchronize the massive parameters, due to the limited Ethernet bandwidth. Transmitting all the parameters through Ethernet will dramatically reduce the training efficiency. Besides, due to the long transmitting time, GPUs have to wait for the data transmitted among servers without doing any computation. Some groups connect the servers with 40G-Ethernet to shorten the transmitting time, but the devices are unusual for most of the researching groups. Anyway, the idle GPU time can not be neglected. To overcome this problem, Adam Coates et al. [4] proposed a model parallelism structure using 16 distributed GPUs. But the structure is deeply customized that is not viable in a reasonable amount of time unless the system also supports data parallelism, as discussed in [3]. GeePS improved the parameter center with GPU caches to increase the throughput [6]. But, it does not reduce the size of parameters transmitted over Ethernet.

To address the above problems, we propose a novel hybrid parallelism framework 'Wheel' to accelerate the CNN training by reducing the size of the transmitted data and fully using GPUs simultaneously. As shown in Figure 1, Wheel partitions the flows in data parallelism. Keeping the extensibility from data parallelism, Wheel obtains higher efficiency. Specifically, Wheel contains many workers to train CNN with different input data and manages the parameters by introducing a parameter center that is a standard architecture in data parallelism. Besides, the CNN layers are partitioned into convolutional module and fully-connected module. They are set in many workers while the fully-connected workers should run in the server on which the parameter center is deployed. Thus most of the data are transmitted within the server and only a few data are transmitted among servers. The communication time transmitting data is significantly reduced. On the other hand, to fully run each GPU and reduce the idle time, Wheel deploys multiple workers on each GPU. While one worker is waiting for data, another one in the same GPU executes its computing task. The workers in each GPU run concurrently and repeatedly like Wheels. Each GPU can do more computations to accelerate the training process. To our best knowledge, it is the first time to reduce the size of the transmitted data and fully utilize GPUs simultaneously. Experiments are conducted to show the performance of the proposed scheme with comparison to the popular parallel CNN architectures.

The main contributions of this papers are:

(1) A simple yet effective hybrid parallelism is devised. It eliminates the large bandwidth requirements of data parallelism using distributed GPUs. Moreover, its simple structure brings high adaptability for various distributed environments outperforming model parallelism.
(2) Alternate strategy is proposed to increase GPU usage. Each of the GPUs are set with multiple CNN workers. Each of the workers in the same GPU run alternately. While one worker is suspended, there is alway another one is ready to run. That avoids the effects from inevitable data communications.
(3) A novel framework 'Wheel' is proposed and implemented. With hybrid parallelism and alternate strategy, Wheel obliterates the bandwidth limit using distributed GPUs to train CNN efficiently. The experiments present the salient speedup via Gigabit Ethernet compared with other distributed frameworks.

2 RELATED WORK

Deep learning methods have been widely used for various multimedia and computer vision tasks, such as image classification, visual representation [15], image retrieval [7], etc. Convolutional neural networks (CNNs), as the most famous deep networks, have attracted much attention since Alexnet [17] won the champion with top-1 and top-5 error rates of 39.7% and 18.9% in ILSVRC2010. Since then, various CNN were proposed. VGG [24] contains more than 16 layers with more parameters than Alexnet. GoogLeNet [26] increases the number of layers and sets a local MLP [19]. ResNet [12] puts the training goal on residuals to deepen the networks to more than 100 layers. With the hundreds layers, ResNet won ILSVRC2015 with top-5 error-rate of 3.5%. Nonetheless, the parameters and the intermediate data of these CNN models are always over tens of millions.

Therefore, to train an effective model in shorter time, optimizing the implementations is significant. To train an effective model with efficient approaches, Peisong Wang et al. [29] proposed to increase the running efficiency in term of the low-rank approximation which can be used in test phrases. Besides, Basic Linear Algebra Subprograms (BLAS), the matrix computing library, is used to optimize all the operations in CNNs. Thus, the popular tools such as Caffe [13] and Torch7 [5] use BLAS to obtain ultimate efficiency. Even so, compared with the specific devices, they are much slower.

There are many device providing batch processing, e.g. GPU, FPGA [22, 23] and TPU. With the help of these devices, training CNN can be more efficient. GPU is the most famous one because of cuda. Cuda is a c++ extended library to ease the programming on GPUs. Moreover, GPU devices have

https://cloudplatform.googleblog.com/2016/05/Google-supercharges-machine-learning-tasks-with-custom-chip.html

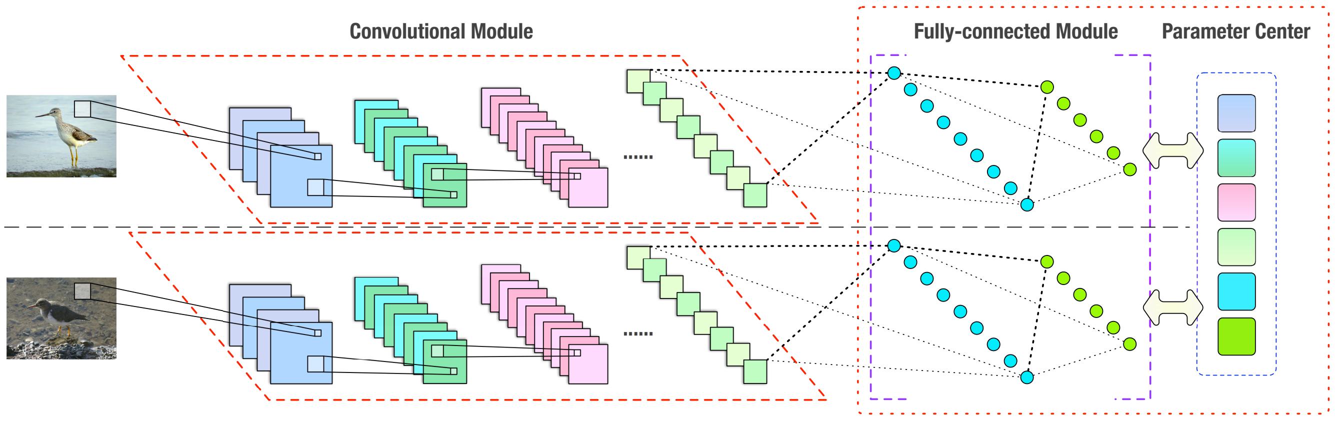

Figure 1: Structure of Wheel.

been being upgraded rapidly. They benefit CNN training day after day. Thus the typical CNN tools, including Caffe [13], Mxnet [2], Torch7 [5], TensorFlow [1], etc., have been GPU supported. Caffe [13] is the fastest one of them, since Caffe is implemented by C++ language and it is perfectly composed of cuDNN which is the fastest customized cuda library for CNN.

To further increase the efficiency, multiple servers and multiple GPUs are used to run in parallelism. Alex Krizhevsky [32] mentioned that the parallel algorithms could be roughly categorized as model parallelism and data parallelism. Model parallelism customizes the running architectures, while data parallelism keeps the model structure and runs them concurrently. Thus model parallelism is alway faster than data parallelism but it is harder to extend. Since data parallelism can be simply extended without any modification on the models, it is widely used in parallel CNN implementations. Convnets [31] ran CNN on two GPUs and four GPUs respectively and illustrated that in the same environment. Its results demonstrate that data parallelism performed worse than model parallelism. Therefore, by improving a CNN tool Theano [28], the approaches proposed to run CNN in multiple GPUs automatically use the data parallelism [8, 21]. It is notable that data parallelism contains a parameter center to synchronize the parameters by synchronous stochastic gradient descent algorithm(ASGD) [26]. The parameter center restricts the extensibility of data parallelism.

The parallelism ideas are applicable to the implementations on distributed servers. SINGA [30] proposed an architecture to construct a distributed cluster for deep learning. DistBelief [9] is a platform to schedule resources integrally. It was used to train Googletnet [26]. But the work recommended using several GPUs instead for less training time. But the communications among multiple GPUs do affect the running efficiency. The effect is more obvious while using distributed GPUs. COTS HPC system [4] used 16 GPUs to train a model. Since the GPUs could not be deployed on single server, COTS HPC system customized an training architecture to make the GPUs more cooperative. The architecture is typically based on model parallelism and it is weakly extensible. as discussed in [3]. Thus‘ it is better to use data parallelism. However, the data transmitted over Ethernet should be well considered.

Using data parallelism, synchronizing parameters with the parameter center among multiple servers is the most significant problem. Some approaches such as [11, 14] integrates the GPUs as middlewares, but the data over Ethernet are still a problem. GeePS [6] proposes to store the data in cache and process them before transmitting. But the size of parameters to be synchronized over Ethernet is not reduced in fact. Actually, according to the characteristics the convolutional layers and fully connected layers can be treated as two parts separately as demonstrated in [3, 16]. The running gaps caused by waiting data can be overlapped such as the approach proposed in [20]. Integrating the above approaches, Wheel is proposed to be an easily extensible parallel architecture on distributed GPUs. The next section describes our Wheel for CNN acceleration.

3 WHEEL

We propose a novel architecture named 'Wheel' to accelerate CNNs using distributed GPUs deployed on multiple servers. Wheel mainly consists of a hybrid parallelization scheme synthesizing the high adaptability of data parallelism and the high efficiency of model parallelism, and a scheduling strategy running workers alternately. Specifically, the hybrid parallelism of Wheel extremely decreases the bandwidth consumption. Most data are transmitted inside the server but only a few are transmitted through Ethernet. Thus Wheel does not rely on high efficient Ethernet and can be used in common Gigabit Ethernet. Besides, the alternate strategy of Wheel fulfills any of the GPUs with multiple workers. The workers in the same GPU are of the same type. They can run alternately and would never conflict. With the alternate strategy, the key computing resources GPUs would never be idle. Thus Wheel accelerates the CNN training processes to an exciting speed. We will elaborate them in the following subsections.

3.1 Hybrid Parallelism

Wheel uses hybrid parallelism synthesizing the high adaptability of data parallelism and the high efficiency of model parallelism. Generally, Wheel consists of several groups of workers to run CNN models. Each of the groups serves as an entire CNN model. Assigning different input data, the groups of workers do their forward and backward propagations concurrently. After each running iteration, the groups synchronize their parameters and gradients with the parameter center. That is much similar to data parallelism which is adaptable for various clusters. However, data parallelism would consume much time to synchronize parameters with parameter center due to the millions even billions of parameters. To avoid transmitting such large parameters, hybrid parallelism is devised to reorganize the workers.

Focusing the widely used and well performed CNN Alexnet [17] and VGG [24], they can be roughly partitioned to two modules: fully-connected module and convolutional module, as shown in Figure 1. Fully-connected module contains all the fully-connected and their upper layers in CNN (i.e., the fc6, fc7, fc8 and softmax layers in Alexnet). Convolutional module contains all the layers below fully-connected layers (i.e., the convolutional layers, pooling layers and normalization layers in Alexnet). Actually, this partition is a typical CNN structure: convolutional module for feature extraction and fully-connected module for semantic comprehension. Wheel defines two kinds of workers running convolutional module and fully-connected module separately. Each convolutional worker corresponds to one certain fully-connected worker. Thus the number of fully-connected workers is as the same as the number of convolutional workers. Especially, all the fully-connected workers must be deployed on the same server with the parameter center.

In this setting, there are three kinds of communicated data among the modules: 1) the outputs with labels and residuals between convolutional module and fully-connected module, 2) the convolutional parameters between convolutional module and the parameter center, 3) the fully-connected parameters between fully-connected module and the parameter center. These transmissions are preseted in Figure 2. The small data numbered 1 to 5 are transmitted over Ethernet and the amount of them. the large data numbered 6 and 7 among fully-connected workers and the parameter center are transmitted inside the servers, since fully-connected workers and the parameter center are deployed on the same server in Wheel. With hybrid parallelism, Wheel effectively reduces the data to be transmitted over Ethernet. More details are discussed in Section 3.4.

3.2 Alternate Strategy

Hybrid parallelism partitions the CNN model and sets the convolutional module and fully-connected module in different workers. The workers may be deployed in different GPUs

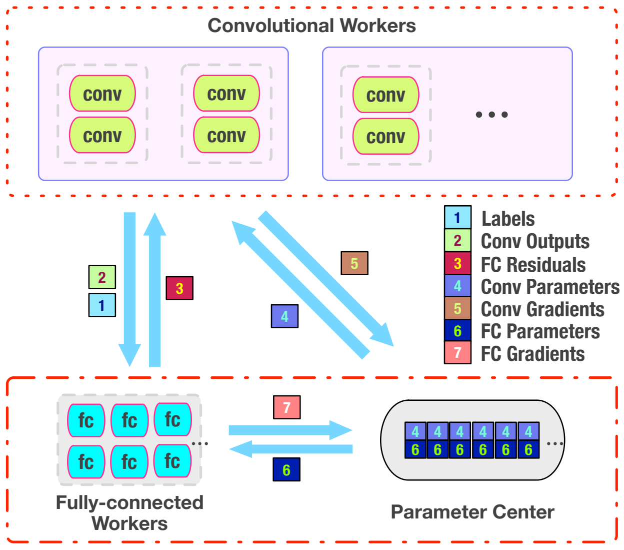

Figure 2: Communications with FC Module.

even in different servers. After the convolutional has done its forward propagation, it has to wait for the residuals from fully-connected workers. Similarly, the fully-connected workers may wait for the output from convolutional workers. While the workers waiting for necessary data, the processes are suspended and the GPUs go idle. This is different from reading input data asynchronously. Reading data is irrelevant to training process, but the data from remote workers are necessary for the next computations. Thus the transmitting speed over Ethernet affect the GPU idle time. GPUs are the key computing resources. The idle GPU time explicitly indicates the loss of efficiency. To reduce or eliminate the idle GPU time, we set some other workers to these idle GPUs. In Wheel, we use Alternate Strategy to better make use of the GPUs.

Alternate strategy is to make GPUs run multiple workers alternately. Here we put some workers with same type in each of GPUs to avoid their conflicts. The workers in one GPU would run alternately. They would halt and receive the remote data in a certain same time period. The same period ensures that the workers in one GPU would execute periodically. A typical running period is presented in Figure 3.

Figure 3 presents two convolutional workers named Conv 0 and Conv 1 in the same GPU run with alternate strategy. At the beginning, Conv 1 is suspended for the residuals from fully-connected module and Conv 0 starts to do forward propagation. At the moment Conv 0 finishes its propagation, Conv 1 has prepared all the necessary data and starts to do backward propagation immediately. After it has done the backward propagation, the data for next forward propagation are ready. Conv 1 does its forward propagation continuously. Then the two workers run alternately. Notably, input data, residuals and new parameters are loaded asynchronously. That do not affect the execution of main flow.

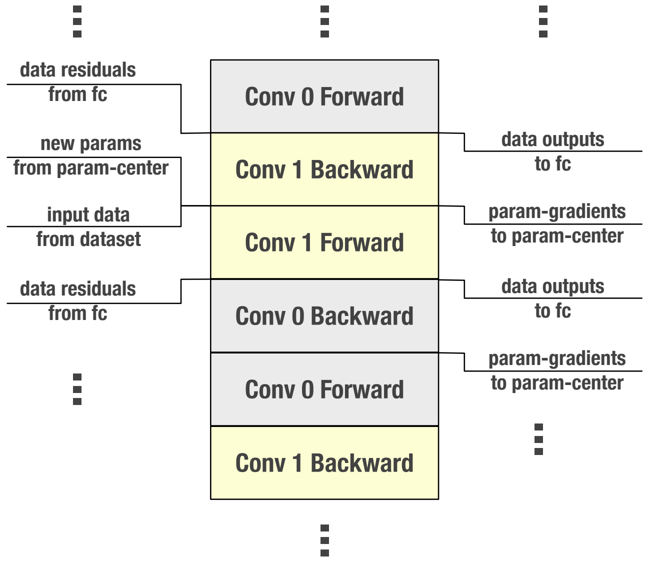

Figure 3: Procedure of Convolutional Module.

The same situation appears in the GPU with multiple fully-connected workers. Since the fully-connected workers would finish its tasks in much shorter time, many fully-connected workers would be set in one GPU. They run alternately to fulfill the GPU computing resources. More details on the number of workers would be discussed in 3.4.

3.3 Simulation

Here we demonstrate the collaborative processes in several training periods of Wheel with three GPUs. We set 4 convolutional workers Conv 0, Conv 1, Conv 2 and Conv 3 in GPU 1 and GPU 2 with two workers in each GPU, as shown in Figure 4. We set the related four fully-connected workers (whose id are signed in the block) in GPU 0. Since the four convolutional workers could not fulfill the GPU 0, we set the parameter center in GPU 0 either. We mark the processes run in GPUs as Conv Forward Propagation, Conv Backward Propagation, FC Propagation (including forward and backward propagations), and Parameters Updating (in parameter center). The widths of the processes indicate the running time. Obviously Conv Backward Propagation takes the longest time while Parameters Updating is the fastest one. Other operations (data reading, parameter synchronizing, etc.) in training are preprocessed by CPUs in parallel, thus they are not noted here.

The simulation training Alexnet [17] with Wheel is presented in Figure 4. Before Conv 0 starts to do its forward propagation, the parameters have already received from the parameter center. When Conv 0 has done its forward propagation at moment 1, Conv 0 sends its output data to FC 0. At that time, Conv 3 has been running in GPU 2, FC 0 starts to run in GPU 0, and Conv 1 starts to run. GPU 1 runs Conv 1 alternately. At moment 2, Conv 1 has done its iteration, and Conv 0 is awaken again. At that time, FC 0 has already done its propagation and has sent the residuals to Conv 0. At that moment, Conv 0 starts its forward propagation. When it has finished its iteration, at moment 3, it sends its gradients to the parameter center. Since the gradients from FC 0 have already been in the parameter center, the gradients of the CNN composed of Conv 0 and FC 0 are complete. Then the parameter center starts to update the parameters. The same situation appears at moment 5. When Conv 3 has done its iteration, the parameter center updates the parameters again. All the convolutional workers process following the same flow, and we will find that there are a small amount of idle time in the GPUs except GPU 0 where fully-connected workers and the parameter center are set in.

3.4 Discussion

In this subsection, we address the details of Wheel and discuss them qualitatively. Wheel is an extensible and effective framework based on hybrid parallelism. To use it well, we need know more about the parallel structure, memory usages and network usages.

3.4.1 Parallel Structure. In Wheel, the convolutional workers and the fully-connected workers are strictly corresponded. Connecting the corresponding workers, several groups of CNN models are got. These groups run in data parallelism and using ASGD to synchronize their parameters. Therefore, Wheel inherits the advantages of data parallelism which can be easily extended.

To use the idle GPU time, each GPU needs at least two workers in it to keep it running alternately. To avoid unexpected conflicts in running process, it is recommended that the workers in the same GPU should be the same kind of workers that can unify the running period. Since convolutional module costs much longer time than fully-connected module, we would use more GPUs for convolutional workers to balance the efficiency.

Since the time run fully-connected workers is less than the time run convolutional workers, if there are two convolutional workers in one GPU, the GPU would run over time. We could suppose the periods of the convolutional workers are both $2\times t$ which is t averagely. If there are more than two convolutional workers in one GPU, the periods of the convolutional workers would be $n \times t$. Here n indicates the number of GPUs and $n > 2$. The average CNN period is t that is as the same as setting two workers. Therefore, setting excessive workers in one GPU would not increase the efficiency. If there are k GPUs running convolutional workers, the fully-connected workers have to response every t/k. Setting all the fully-connected workers in one GPU may delay the training process. Consequently, if the fully-connected workers cost p for one response, there should be at least $(p \times k)/t$ GPUs to ensure the average convolutional period.

In most of the time, the number of GPUs hardly organized in the above proportion. To better utilize the computing resources, the GPUs could be adjusted. If the GPU running fully-connected workers is fulfilled, the workers could be set in another GPU. The only constraint is the GPU should be

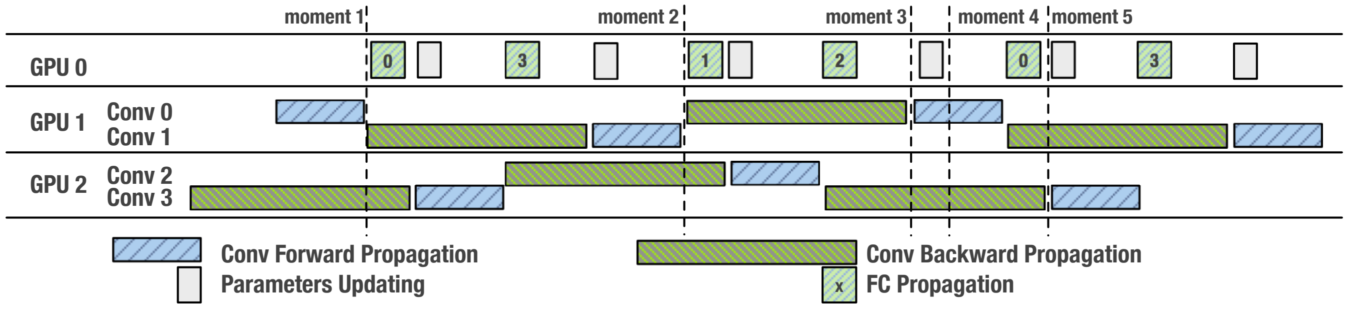

Figure 4: Execution Flow in Wheel

deployed with the parameter center either (to synchronize the parameters inside server). If there are exactly many GPUs, a tree-like strategies could be used for synchronizing multiple parameter centers hierarchically. Conversely, if only a few GPUs are used to train a CNN, we do not need many workers, otherwise redundant operations would lead to unnecessary time-consuming. With a few workers, the GPU in which fully connected workers are set is idle in most time. In that scenario, setting the parameter center in the GPU could utilize the resources more adequately. This is sampled in Figure 4.

Empirically, using Wheel on GPUs in different types is workable. Due to one fully-connected worker is strictly corresponded to one convolutional workers, the transmitted data among them would not be confused. After they got the gradients, the parameter center would synchronize them. Meanwhile, the waiting time would be overlapped. The only problem is, since the GPUs may have different running periods, some conflicts would appear and consume redundant time.

3.4.2 Memory. Since there are many workers set in one GPUs, an intuitive problem is whether there is enough memory for the large-scale parameters. There are four considerations: 1) Current GPU memory is enough for a common CNNs. 2) The partitioned model leads to the partitioned memory usage. 3) It is not necessary to set more than two convolutional workers in one GPU, or there are always at least two waiting workers. 4) The parameters of fully-connected workers would not be loaded concurrently. The parameters are from the parameter center and stored in the main memory until the worker receives the outputs from convolutional worker. Then the parameters are loaded to GPU and used immediately. After that, the parameters are eliminated. Therefore, there would only one set of fully-connected parameters in the GPU. Generally, training common CNNs, Wheel would not be limited by memory.

3.4.3 Network Bandwidth. The hybrid parallelism is proposed to reduce the transmitted data over Ethernet. Its effectiveness should be discussed. As shown in Table 1, using Hybrid Parallelism, there are only about 12.2 million data on Ethernet for one iteration. Using Data Parallelism, there are about 124.8 million data on Ethernet which is about ten times of using Hybrid Parallelism. It means that hybrid parallelism could hold about ten more workers than data parallelism in limited bandwidth environments. Since current Ethernets are always full-duplex in which the data can be upload and download concurrently, the Ten-Gigabit Ethernet can transmit up to 640 million data per seconds. With the factor of the CNN running time (in the following experiments, it is about 1.69 seconds), nearly a hundreds of CNN models can be hold concurrently in this environment. If the workers finally exceed the Ethernet capability, they could be composed in several intranets and linked with some additional parameter centers.

Table 1: Comparison between Data Parallelism and Hybrid Parallelism (Alexnet)

Approach	Data on Ethernet	Size
Data Parallelism	Sample Labels	1K
	Convolutional Parameters	3.7M
	Convolutional Gradients	3.7M
	Fully-Connected Parameters	58.6M
	Fully-Connected Gradients	58.6M
	Total	**124.8M**
Hybrid Parallelism	Sample Labels	1K
	Convolutional Parameters	3.7M
	Convolutional Gradients	3.7M
	Convolutional Outputs	2.4M
	Fully-Connected Residuals	2.4M
	Total	**12.2M**

3.4.4 Adaptability. Wheel performs well for the CNN models containing fully-connected layers, such as Alexnet and VGG, due to the reduction of transmitting data over Ethernet. The CNN models without any fully-connected layers such as ResNet are degenerate to normal data parallelism. But by partitioning these kinds of CNNs technically, they could run in Hybrid Parallelism either. Moreover, Wheel contains only two kinds of workers and one parameter center. The constraints deploying the workers are simple. Thus it can easily adapt various distributed environment.

4 EVALUATION

The experiments are conducted by using four K40c GPUs with ECC off and maximum clock speed. They are set on two Lenovo ThinkStation P900 workstations. The servers are connected directly to construct a Gigabit Ethernet. We take the classical Alexnet as the CNN model and train it on the GPUs with 256 image per batch. The modules are extracted from Caffe. They are reorganized with C++ language and the distributed library 'openMPI'. Based on this configuration, we are going to discuss the performance of Wheel. Since Wheel aims to accelerate the training process, we compare the results with Caffe [13] implemented with cuDNN, the fastest tool in single GPU.

4.1 Performance Improvements

In this subsection, we discuss the performance improvements using Wheel in terms of training time, network flow and GPU usages. Here we set the convolutional module and fully-connected module as described in Section 3.1. We set six fully-connected workers and the parameter centers in one of the four GPUs. And we set six convolutional workers in other three GPUs with two in each GPU. With this configuration, each of the GPUs contains multiple workers to overlap their idle time. It is notable that, in the server with the fully-connected workers, there are two convolutional workers. The communications of them with their corresponded fully-connected workers and the parameter center are inside the server. These data are not transmitted over Ethernet.

We then evaluate the network usages by a network monitor tool 'iptraf'. The up- and down-link speeds of the two servers are both about 42MB per second. Since two of the convolutional workers do not transmit data over Ethernet, the data over Ethernet are about four convolutional workers. However, the transmitted data is much less than the parameters of Alexnet. It confirms that Wheel effectively reduces the transmitted data.

We then evaluate the time for one training iteration, as shown in Figure 5. Caffe costs 0.85 seconds for running one iteration. The middle operations mainly indicate the operations between the forward and backward propagations of convolutional module. In Caffe, they consist of the forward and backward propagagions of fully-connected module. If we partition the model implemented by Caffe and deploy the two modules on two servers, about 1.08 seconds are needed to run an iteration. During the time for the middle operations, the GPU is idle to wait the data over Ethernet. That slows the running process. Using alternate strategy, the time for middle operations is stretched, but during that time, another iteration is running on the same GPU. That is, although the total time running one iteration in a worker is about 1.69 seconds, it costs only about 0.845 seconds per iteration in a GPU with convolutional workers averagely. It is notable that the time computing fully-connected modules seems to be eliminated, but the redundant operations transmitting data 'waste' the time. However, alternate strategy overlap the idle GPU time in practice.

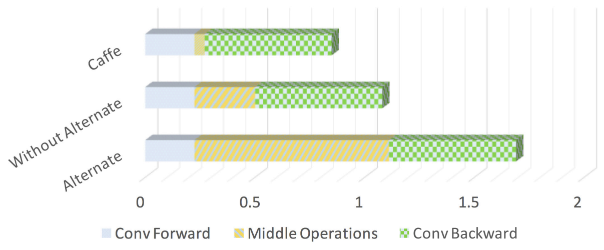

Figure 5: Time for one training iteration.

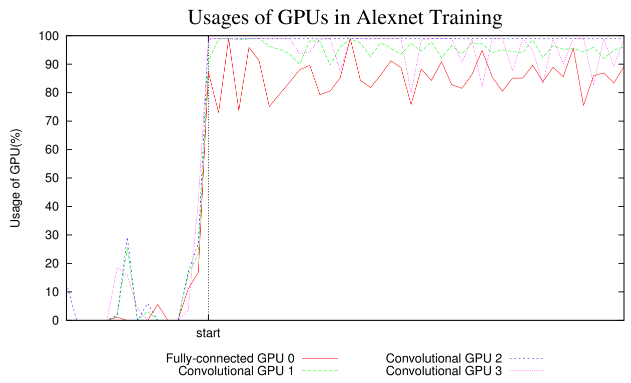

Figure 6: Usages of GPUs in Alex Training.

Another measurement on the effectiveness of Wheel is the usages of GPUs. As shown in Figure 6, when we start to train, the usages of GPUs grow up immediately and most of them are up to 100%. Since there are only six convolutional workers, and the fully-connected workers and the parameter center can not fulfill the computing resources, the usage of GPU 0 is between 80% and 90%. In contrast, the convolutional workers always run, the usages of the GPUs are nearly 100%. Their slight oscillation indicates the GPUs' redundant operaionts. Specifically, the GPU 2 is deployed on the same server of GPU 0. Its communications are faster than other GPUs. Thus it contains rare oscillation.

Consequently, Wheel does reduce the flow over Ethernet and accelerate the training process.

4.2 Availability

To confirm that Wheel is exactly available to train a network and would not downgrades the training performance, we train a Alexnet model to get validation accuracy of 30%. We record the loss curve for training batches and mark the validation accuracies every 10,000 iterations, as shown in Figure 7. Since the curve of ASGD is different from basic SGD, the convergence speed is not strictly related to the number of trained batches. Nonetheless, the number of iterations Wheel can train is three times of Caffe and the time using Wheel is about twice shorter. However, it is confirmed that Wheel can exactly train an efficient CNN model.

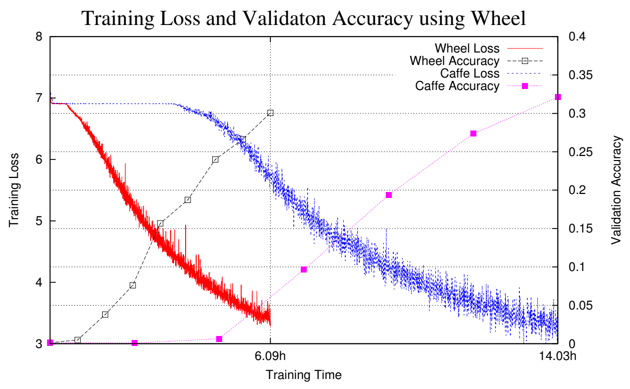

Figure 7: Alexnet Training Process using Wheel

4.3 Acceleration with Multiple GPUs

In this subsection, we evaluate the accelerating performance with multiple GPUs. We evaluate the time training 20 iterations (5,120 images). That is an accepted criterion on evaluating the training efficiency. Since the official measurement is based on GPU typed K40 which is slower than K40c, we take its running time in our experiments as the benchmark.

As shown in Figure 8, in standard Caffe, training 5,120 images costs about 16.84 seconds. If we simply partition the model and set them on two distributed GPUs, the training costs 24.06 seconds. The extra time is used to transmit the data. Using alternate strategy, the time transmitting data is utilized, the time decreases to 16.99 seconds. Since the GPU running fully-connected workers is idle in most time, it seems to be 'wasted'. Even if we set the parameter center in it, the two-GPUs structure seems not as good as Caffe. However, the phenomenon is caused by the aim of Wheel. Wheel is for multiple GPUs and would perform well with more GPUs. With adding another GPU, extra computing resources are appended without any extra consumptions. Thus Wheel using three GPUs costs about half time of using two GPUs. Similarly, Wheel using three GPUs costs about one third time of using two GPUs and using Caffe. It costs 5.48 seconds to train 5,120 images. With the increasing number of GPUs, the performance increases nearly linearly.

4.4 Comparison

Table 2 presents the accelerating ratios of some approaches using multiple GPUs. The results of Theano-based approach and the Convnets are evaluated on the GPUs on single server. They need not consider the effect of Ethernet. Thus theoretically they are faster than GeePS [6] and Wheel on multiple servers. GeePS [6] is a new parameter server GeePS to optimize the parameter synchronizations among multiple servers. Even so, these approaches using four GPUs achieve a speed-up of 2.18 times compared to using Caffe. Even the ideal efficiency of GeePS using four GPUs achieves a speed-up of 2.32 times compared to using Caffe [6]. Wheel is exactly the one using GPUs adequately and achieves a speed-up of 3.07 times. The acceleration of parallel approaches using two GPUs achieves a speed-up of one times compared to Caffe because it is hard to balance the synchronization and parallelism, especially through multiple servers. For the same reason, the Convnets using model parallelism (mp) and hybrid parallelism (hp) those can customize the data flow perform better than using data parallelism (dp). However, using four and more GPUs, with the optimization in Wheel, CNN models would be accelerated in a high ratio.

The criterion is from Caffe official site:
http://caffe.berkeleyvision.org/performance_hardware.html

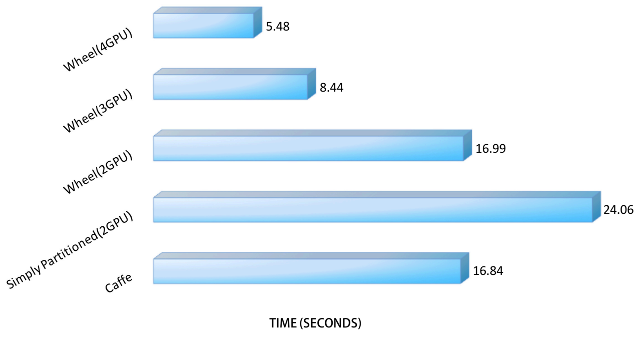

Figure 8: Training time(sec) per 5120 Images.

Table 2: Acceleration compared with Caffe using cuDNN.

Condition	Approach	2-GPUs	4-GPUs
Single Server	Theano-based [10]	0.97×	-
	Convnets [31]	(dp)1.5× (mp)1.59×	(dp)1.45× (hp)2.18×
Multiple Servers	GeePS [6]	1.07×	2.13×
	Wheel	0.99×	**3.07**×

5 CONCLUSION

We propose a novel framework 'Wheel' to accelerate CNN training using distributed GPUs. The hybrid parallelism of Wheel effectively reduces the to-be-transmitted data over Ethernet to adapt lower speed Ethernets and the alternate strategy of Wheel increases the GPU usages. The experiments demonstrate that Wheel does accelerate the training process of Alexnet with distributed GPUs and obtains the better performances with more GPUs. Experiments are conducted to show the outperformance of the proposed scheme over the state-of-the-art parallel approaches.

ACKNOWLEDGMENTS

This work was partially supported by the 973 Program (Project No. 2014CB347600), the National Natural Science Foundation of China (Grant No. 61522203 and 61402228) and the Natural Science Foundation of Jiangsu Province (Grant BK20140058 and BK20170033).

REFERENCES

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, and others. 2016. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467* (2016).

[2] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. 2015. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274* (2015).

[3] Trishul Chilimbi, Yutaka Suzue, Johnson Apacible, and Karthik Kalyanaraman. 2014. Project adam: Building an efficient and scalable deep learning training system. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. 571–582.

[4] Adam Coates, Brody Huval, Tao Wang, David J. Wu, and Andrew Y. Ng. 2013. Deep learning with COTS HPC systems. *Proceedings of the 30 th International Conference on Machine Learning* volume 28 (2013).

[5] Ronan Collobert, Koray Kavukcuoglu, and Clément Farabet. 2011. Torch7: A matlab-like environment for machine learning. In *BigLearn, NIPS Workshop*.

[6] Henggang Cui, Hao Zhang, Gregory R. Ganger, Phillip B. Gibbons, and Eric P. Xing. 2016. GeePS: Scalable deep learning on distributed GPUs with a GPU-specialized parameter server. In *Proceedings of the Eleventh European Conference on Computer Systems*. ACM, 4.

[7] Qi Dai, Jianguo Li, Jingdong Wang, and Yu-Gang Jiang. 2016. Binary optimized hashing. In *Proceedings of the 2016 ACM on Multimedia Conference*. ACM, 1247–1256.

[8] Dipankar Das, Sasikanth Avancha, Dheevatsa Mudigere, Karthikeyan Vaidynathan, Srinivas Sridharan, Dhiraj Kalamkar, Bharat Kaul, and Pradeep Dubey. 2016. Distributed deep learning using synchronous stochastic gradient descent. *arXiv preprint arXiv:1602.06709* (2016).

[9] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Andrew Senior, Paul Tucker, Ke Yang, and Quoc V. Le. 2012. Large scale distributed deep networks. In *Advances in neural information processing systems*. 1223–1231.

[10] Weiguang Ding, Ruoyan Wang, Fei Mao, and Graham Taylor. 2014. Theano-based large-scale visual recognition with multiple gpus. *arXiv preprint arXiv:1412.2302* (2014).

[11] Anshuman Goswami, Jeffrey Young, Karsten Schwan, Naila Farooqui, Ada Gavrilovska, Matthew Wolf, and Greg Eisenhauer. 2016. GPUShare: Fair-Sharing Middleware for GPU Clouds. In *Parallel and Distributed Processing Symposium Workshops, 2016 IEEE International*. IEEE, 1769–1776.

[12] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 770–778.

[13] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. 2014. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the ACM International Conference on Multimedia*. ACM, 675–678.

[14] Atsushi Kawai, Kenji Yasuoka, Kazuyuki Yoshikawa, and Tetsu Narumi. 2012. Distributed-Shared CUDA: Virtualization of large-scale GPU systems for programmability and reliability. In *The Fourth International Conference on Future Computational Technologies and Applications*.

[15] Markus Koskela and Jorma Laaksonen. 2014. Convolutional Network Features for Scene Recognition. In *Proceedings of the ACM International Conference on Multimedia - MM '14*. 1169–1172. https://doi.org/10.1145/2647868.2655024

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

[17] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2012. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*. 1097–1105.

[18] Chuck L Lawson, Richard J. Hanson, David R Kincaid, and Fred T. Krogh. 1979. Basic linear algebra subprograms for Fortran usage. *ACM Transactions on Mathematical Software (TOMS)* 5, 3 (1979), 308–323.

[19] Min Lin, Qiang Chen, and Shuicheng Yan. 2014. Network in network. In *ICLR*.

[20] Min Lin, Shuo Li, Xuan Luo, and Shuicheng Yan. 2014. Purine: A bi-graph based deep learning framework. *arXiv preprint arXiv:1412.6249* (2014).

[21] He Ma, Fei Mao, and Graham W. Taylor. 2016. Theano-MPI: a Theano-based Distributed Training Framework. *arXiv preprint arXiv:1605.08325* (2016).

[22] Garrick Orchard, Jacob G Martin, R Jacob Vogelstein, and Ralph Etienne-Cummings. 2013. Fast neuromimetic object recognition using FPGA outperforms GPU implementations. *IEEE transactions on neural networks and learning systems* 24, 8 (2013), 1239–1252.

[23] Markos Papadonikolakis and Christos-Savvas Bouganis. 2012. Novel cascade FPGA accelerator for support vector machines classification. *IEEE transactions on neural networks and learning systems* 23, 7 (2012), 1040–1052.

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

[25] Ilya Sutskever, James Martens, George E Dahl, and Geoffrey E Hinton. 2013. On the importance of initialization and momentum in deep learning. *ICML (3)* 28 (2013), 1139–1147.

[26] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. 2015. Going deeper with convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 1–9.

[27] Jinhui Tang, Xiangbo Shu, Zechao Li, Guo-Jun Qi, and Jingdong Wang. 2016. Generalized deep transfer networks for knowledge propagation in heterogeneous domains. *ACM Transactions on Multimedia Computing, Communications, and Applications* 12, 4s (2016), 68.

[28] The Theano Development Team, Rami Al-Rfou, Guillaume Alain, Amjad Almahairi, Christof Angermueller, Dzmitry Bahdanau, Nicolas Ballas, Frédéric Bastien, Justin Bayer, Anatoly Belikov, and others. 2016. Theano: A Python framework for fast computation of mathematical expressions. *arXiv preprint arXiv:1605.02688* (2016).

[29] Peisong Wang and Jian Cheng. 2016. Accelerating convolutional neural networks for mobile applications. In *Proceedings of the 2016 ACM on Multimedia Conference*. ACM, 541–545.

[30] Wei Wang, Gang Chen, Anh Tien Tuan Dinh, Jinyang Gao, Beng Chin Ooi, Kian-Lee Tan, and Sheng Wang. 2015. SINGA: Putting deep learning in the hands of multimedia users. In *Proceedings of the 23rd ACM international conference on Multimedia*. ACM, 25–34.

[31] Omry Yadan, Keith Adams, Yaniv Taigman, and MarcAurelio Ranzato. 2013. Multi-gpu training of convnets. *arXiv preprint arXiv:1312.5853* 9 (2013).

[32] Matthew D. Zeiler and Rob Fergus. 2014. Visualizing and understanding convolutional networks. In *European Conference on Computer Vision*. Springer, 818–833.