Involving CPUs into Multi-GPU Deep Learning

Tung D. Le
IBM Research - Tokyo
Tokyo, Japan
tung@jp.ibm.com

Taro Sekiyama
IBM Research - Tokyo
Tokyo, Japan
sekiym@jp.ibm.com

Yasushi Negishi
IBM Research - Tokyo
Tokyo, Japan
negishi@jp.ibm.com

Haruki Imai
IBM Research - Tokyo
Tokyo, Japan
imaihal@jp.ibm.com

Kiyokuni Kawachiya
IBM Research - Tokyo
Tokyo, Japan
kawatiya@jp.ibm.com

ABSTRACT

The most important part of deep learning is training a neural network, which often processes a large amount of data and takes days to complete. Data parallelism is widely used for training a deep neural network on multiple GPUs in a single machine thanks to its simplicity. Nevertheless, its scalability is bound by data transfers that are mainly for exchanging and accumulating gradients among the GPUs. In this paper, we first propose a novel approach to data parallel training, called CPU-GPU data parallel training, which utilizes free CPUs on host to speed up the training in GPUs. Next, a cost model is proposed to analyze the performance of both the original and CPU-GPU data parallel trainings. By using the cost model, we formally show why our approach is better than the original one and what are the remaining obstacles. Finally, we further optimize the CPU-GPU data parallel training by introducing chunks of layers and proposing a run-time algorithm that automatically finds the best configuration for the training. The algorithm is effective for very deep neural networks that are a trend in deep learning. Experimental results showed that we achieved speedups of 1.19, 1.04, 1.21, 1.07 for four state-of-the-art neural networks: AlexNet, GoogLeNet-v1, VGGNet-16, and Resnet-152, respectively. Weak scaling efficiency greater than 90% was achieved for all networks across four GPUs.

CCS CONCEPTS

• **Software and its engineering → Software performance**;

KEYWORDS

Deep learning, data parallelism, GPUs, CPUs

ACM Reference Format:
Tung D. Le, Taro Sekiyama, Yasushi Negishi, Haruki Imai, and Kiyokuni Kawachiya. 2018. Involving CPUs into Multi-GPU Deep Learning. In *Proceedings of the 9th ACM/SPEC International Conference on Performance Engineering (ICPE'18).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Deep learning is an effective tool for solving complex signal processing problems such as ones in computer vision, speech recognition, and natural language processing. It has been successfully used to recognize objects in digital images. In 2012, a deep convolutional neural network called AlexNet [17] achieved outstanding image classification results in the ILSVRC-2012 competition with a top-5 test error rate of 15.3%. In 2015, rectifier neural networks surpassed human-level performance on image classification with 4.94% top-5 error [12]. Variations of the deep neural networks for image recognition were also used for the detection of pulmonary nodules in the analysis of lung cancer [4]. LSTM (Long Short Term Memory) networks made a major milestone in conversational speech recognition in which a new industry record of 5.5% word error rate has been reached [20].

A deep neural network has many layers (the deepest one up to date is the 1001-layer Resnet [13]) and is trained with a big dataset. Training is mainly based on matrix multiplications and is therefore often accelerated by GPUs. To fully utilize multiple GPUs for training, data parallelism is often used because 1) it is simple to adapt an existing single GPU training for multiple GPUs with small extension and 2) it fully utilizes the GPU-aware optimized training in a single GPU. In data parallel training, every GPU has the same neural network that will be trained with different inputs. Once they have finished the forward and backward phases of one training iteration, a *server* GPU accumulates partial gradients from the other GPUs and updates learnable parameters. The updated parameters on the server GPU are broadcasted to the other GPUs at the beginning of the next training iteration to ensure that every GPU has the same parameters.

However, the scalability of data parallel training is limited by the accumulation and broadcast of gradients. The more GPUs are used, the more data are exchanged among GPUs. A simple yet effective solution is using a tree layout of GPUs so that some communications are done in parallel, which is used in deep learning frameworks: BVLC/Caffe [15] and Torch [2]. Another approach is using topology-aware communication libraries such as NCCL [3]. TensorFlow framework [5] does both gradient accumulation and parameter update synchronously on CPUs at the end of the backward phase, which is flexible for distributed training but slow for training on a single machine. MXNet [6] provides a mix of the above strategies such as gradient accumulation is done on CPUs while parameter update is done on GPUs.

In this paper, we first propose a novel approach to data parallel training, named *CPU-GPU data parallel (CGDP) training*, which utilizes free CPUs on host to speed up the training in GPUs. In this approach, gradients are collected and accumulated on host

ICPE'18, April 2018, Berlin, Germany
2018. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

layer-by-layer during the backward phase. Once a partial gradient of a layer is available in a GPU, it is sent to the host. Gradient accumulation is done by CPUs while the GPU still computes partial gradients for the other layers. The accumulated gradient on host is then sent back to the GPU for updating learnable parameters. This approach is particularly effective for convolutional neural networks that are widely used in image processing. Convolutional neural networks usually start with convolutional layers having a small number of parameters and end with fully connected layers having a large number of parameters. Since backward computations are performed from the ending layer to the starting layer, collection and accumulation of the gradients of the ending layers will have been completed by the end of the backward phase even though they might take time. Furthermore, since collecting and accumulating the gradients of the starting layers take less time, they will be completed immediately after the backward phase with a very low overhead.

Second, we propose a cost model for analyzing the performance of data parallel training on multiple GPUs. The model takes into account not only the cost for computation and communication, but also the cost for synchronization among GPU streams that is important to GPU applications. By using the model, we show that the CGDP training is better than the original training for deep neural networks having a small number of layers (less than 30). To the best of our knowledge, this is the first time a cost model for data parallel training on multiple GPUs is proposed.

Finally, we extend the CGDP training by using chunks of layers to deal with very deep and "flat" neural networks in which the number of parameters in a layer is small and roughly equal to the ones in the other layers. For such networks, the cost model shows that the synchronization among GPU streams becomes a bottleneck and makes the training slow, and the purpose of using chunks is to reduce the effect of synchronization on training. We propose a run-time algorithm to automatically determines synchronization points during some first training iterations so that the running time is optimized.

We implemented our idea in the BVLC/Caffe [15] deep learning framework, which is widely used in the deep learning community. Experiments were done with the ImageNet dataset [19] (ILSVRC2012 challenge, 1.2 millions images classified into 1000 categories) on an IBM POWER8 machine coupled with four NVIDIA Tesla P100 GPUs and fast NVLinks among the CPUs and GPUs [1]. Our results showed that the overhead time for gradient accumulation and broadcast at the end of the backward phase was decreased from tens of milliseconds to tens of microseconds for three state-of-the-art convolutional neural networks: AlexNet [17], GoogLeNet-v1 [23], and 16-layer VGGNet (model D) (VGGNet-16 hereafter) [21]. Speedups of 1.19, 1.04, 1.21 were for the three neural networks, respectively, and we achieved more than 90% weak scaling efficiency for all three networks when four GPUs were used. For a very deep neural network, 152-layer Resnet [11] (Resnet-152 hereafter), while the naive CGDP training was slower than the original data parallel training, using chunks and the run-time algorithm made it 1.07 times faster than the original data parallel training. In addition, our approach reduced the training time for AlexNet to reach 50% accuracy from 79 minutes to 62 minutes. Offloading the gradient accumulation onto host helped train VGGNet-16 with 103 images instead of 94 images for each iteration, thanks to more available memory in the GPUs.

The rest of the paper is organized as follows. Section 2 reviews data parallelism for deep learning. Section 3 describes in detail our CPU-GPU data parallel training on a single machine coupled with multiple GPUs. A cost model is presented in the section. Section 4 presents a variant of the CGDP training that uses chunks of layers in training. Then we show a run-time algorithm to automatically optimize the CGDP training with chunks for very deep neural networks. Section 5 shows experiment results for a real dataset, ImageNet [19]. Section 6 discusses related work. Section 7 concludes and summarizes key points in the paper.

2 DATA PARALLELISM FOR DEEP LEARNING

This section briefly reviews the training phase of a neural network using the back-propagation algorithm [10] and its data parallelism.

2.1 Training Deep Neural Networks

We give an overview of training deep neural networks via *feed-forward neural networks* (FFN) that are a fundamental architecture for convolutional neural networks.

The goal of an FFN is to approximate a function f^*, i.e. $y^* = f^*(x)$ maps an input x (a tensor) to a category y^* (a scalar value). In general, an FFN defines a mapping $y = f_\theta(x)$ where the parameter θ is learnt to produce the best function approximation for f^*. If y^* is given in training, we have a supervised training, otherwise, an unsupervised training. In this paper, we focus on supervised training only. A *cost function*, also often called a *loss function*, defines how well f_θ approximates f^*. For example, the *mean squared error* (MSE) loss function is defined on the whole training set $\mathbb{X}$ of N elements as follows:

$$J_\theta = \frac{1}{N}\sum_{x\in\mathbb{X}}(y^* - f_\theta(x))^2$$

Feed-forward neural networks are presented by a composition of many functions, i.e. $f(x) = f^3(f^2(f^1(x)))$ where f^i is the i-th layer of the network. Generally speaking, an l-layer FFN is represented as

$$f_\theta(x) = f^l_{\theta_l}(f^{l-1}_{\theta_{l-1}}(\cdots(f^1_{\theta_1}(x))\cdots))$$

where θ is a set of the layer parameters $\{\theta_1, \theta_2, \ldots, \theta_l\}$.

A layer k is often defined by an activation function to make neural networks nonlinear. Let y_{k-1} be an input vector (the output of the previous layer) of the layer k, an output vector y_k of the layer k is computed as

$$y_k = f^k_{\theta_k} = \sigma(x_k)$$
$$x_k = w_k^T y_{k-1} + b_k$$

where σ is an activation function (e.g., the *rectified linear unit* defined by $\sigma(z) = \max\{0, z\}$) and w_k^T is the transpose matrix of the *weight matrix* w_k. Let m be the size of y_{k-1} and n be the size of y_k. Then, the size of the matrix w_k is $m \times n$. Vector b_k is a *bias vector* of the layer k and has the size of n. In summary, a layer is parameterized by two learnable parameters: the weight matrix w and the bias vector b. In other words, $\theta_k = \{w_k, b_k\}$. Note that, although the equation for x_k is actually for a fully connected layer, the definition of layer can be applied to other kinds such as convolutional layers.

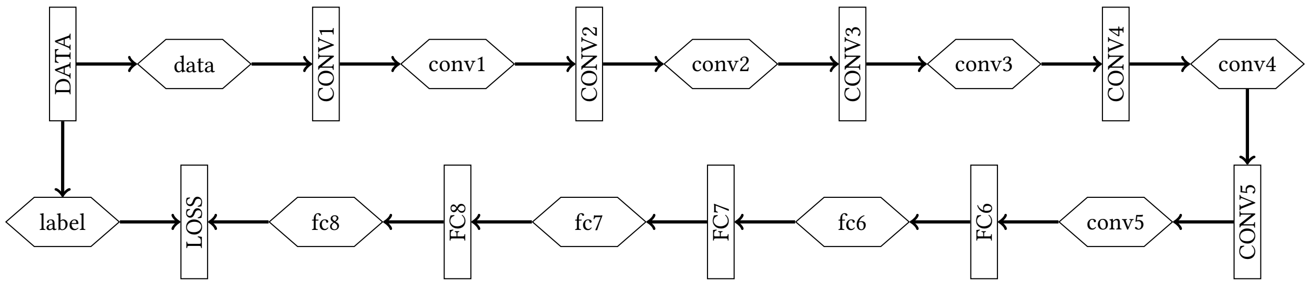


Figure 1: AlexNet deep neural network.

Figure 1 shows an example of AlexNet [17], one of the first practical neural networks. Only layers with learnable parameters are shown. AlexNet comprises five convolutional layers and three fully connected layers. The convolutional layers are represented in the figure by rectangles with "CONV" as a prefix. The fully connected layers are represented by rectangles with "FC" as a prefix. "DATA" represents the input layer. "LOSS" represents the ending layer served as the loss function. Inputs/outputs of a layer are represented by hexagons. Taking the layer "CONV2" as an example, it accepts "conv1" as input and outputs "conv2". Data "conv2" then becomes an input for the layer "CONV3". Data "label" plays a role of truth value, say, the value of y^* in the definition of an FFN.

Training FFNs is almost always based on using gradients with respect to learnable parameters to descend the loss function. In general, it consists of three phases: *forward*, *backward* and *update*. Given an l-layer FFN with learnable parameters $\theta = \{\theta_1, \theta_2, \ldots, \theta_l\}$, the forward phase is to compute a scalar value J_θ. The backward phase computes the gradients of the loss function with respect to the learnable parameters that are $\nabla_{\theta_1} J, \nabla_{\theta_2} J, \ldots, \nabla_{\theta_l} J$. Actually, we need to compute two gradients $\nabla_{w_k} J, \nabla_{b_k} J$ for a layer k. The update phase updates learnable parameters using their gradients in the direction that minimizes the value of the loss function, for example,

$$w_k = w_k - \eta \nabla_{w_k} J$$
$$b_k = b_k - \eta \nabla_{b_k} J$$

where η is a given learning rate.

To train an FFN with a big training set, a minibatch stochastic gradient decent (SGD) algorithm is used. The training is performed through iterations, where, for each iteration, a minibatch (subset) of examples extracted from the training set is used as input.

2.2 Back-propagation Algorithm to Compute Gradients

We briefly review the back-propagation algorithm for gradient computation in the backward phase. The back-propagation is based on the chain rule of calculus that is used to compute the derivatives of composed functions by propagating the derivative information from the loss function back through the composed functions. The back-propagation algorithm is described as follows:

Step B-1: Compute the gradient on the output layer:

$$\nabla_{y_l} J = \nabla_f J_\theta(y^*, f)$$

Step B-2: Compute the gradient on the activation on the l-th layer:

$$\nabla_{x_l} J = \left(\frac{\partial y_l}{\partial x_l}\right)^T \nabla_{y_l} J$$

Step B-3: Compute the gradients on the learnable parameters (weights and biases):

$$\nabla_{w_l} J = \left(\frac{\partial x_l}{\partial w_l}\right)^T \nabla_{x_l} J$$
$$\nabla_{b_l} J = \left(\frac{\partial x_l}{\partial b_l}\right)^T \nabla_{x_l} J$$

Step B-4: Propagate the gradients with respect to the activations of the lower-level layers (e.g., layers with smaller indices):

$$\nabla_{y_{l-1}} J = \left(\frac{\partial x_l}{\partial y_{l-1}}\right)^T \nabla_{x_l} J.$$

Step B-5: Continue steps 2 to 4 until l reaches 1.

where $\left(\frac{\partial y}{\partial x}\right)$ denotes the $m \times n$ Jacobian matrix of a function g for $y = g(x)$, m is the size of the vector y and n is the size of the vector x.

2.3 Typical Data Parallel Training

In this paper, we focus on data parallel training on a single machine coupled with multiple GPUs. Let G be the number of GPUs in the machine. For data parallel training, the GPUs keep the same neural network and train the network with different minibatches. One iteration of data parallel training is as follows:

Step T-1: GPU 0 broadcasts the values of its learnable parameters to the other GPUs. The other GPUs set the values of their learnable parameters to the values from the GPU 0.

Step T-2: Every GPU reads one minibatch and performs the forward phase.

Step T-3: Every GPU performs the backward phase to compute the gradients with respect to its learnable parameters. Let $\nabla^j_{w_k} J$ and $\nabla^j_{b_k} J$ be the gradients with respect to the learnable parameters $\{w_k, b_k\}$ for a layer k computed by a GPU j.

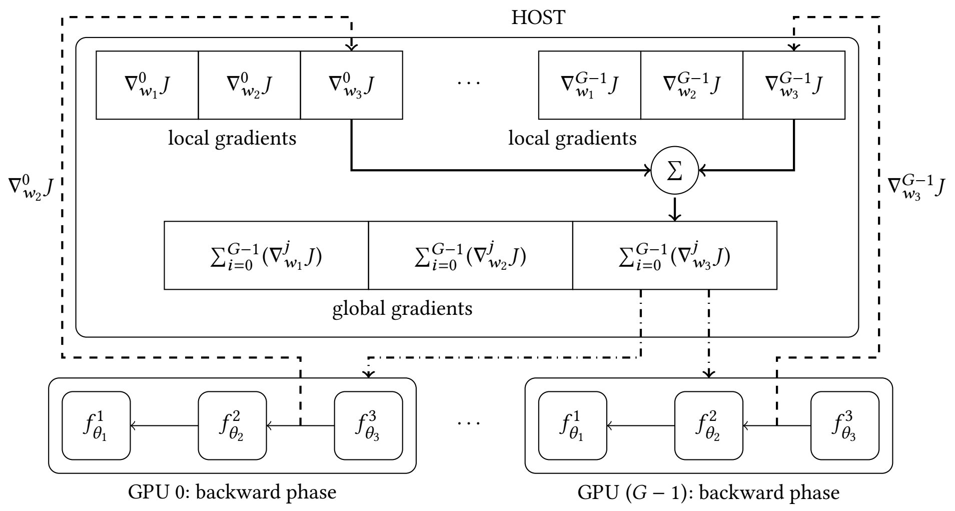


Figure 2: Communication pattern between host and GPUs during the backward phase in the CGDP training.

Step T-4: GPU 0 accumulates the gradients from the other GPUs and computes the mean values of the gradients:

$$\nabla^0_{w_k}J = \frac{1}{G}\sum_{j=0}^{G-1}(\nabla^j_{w_k}J)$$

$$\nabla^0_{b_k}J = \frac{1}{G}\sum_{j=0}^{G-1}(\nabla^j_{b_k}J)$$

Step T-5: GPU 0 updates its learnable parameters by using the computed gradients.

In the above algorithm, GPU 0 plays a role of a parameter server, collecting gradients and updating learnable parameters.

3 CPU-GPU DATA PARALLEL TRAINING

The drawback of the original data parallel training lies on communication steps: Step T-1 and T-4 (Section 2.3), leading to a poor scalability. The two steps depend on the communication pattern among GPUs. Our goal of optimizing data parallel training is to reduce as much as possible the impact of the communication steps on the running time of one training iteration.

3.1 Algorithm

Our algorithm is based on two observations. The first observation is that the gradients for one layer once computed will remain unchanged during the backward phase. Hence, there is no need to postpone gradient accumulation until the end of the backward phase. The second observation is that gradient accumulation can be performed with the support of CPUs on host. Based on these two observations, we make the steps T-1 (parameter broadcast) and T-4 (gradient accumulation) overlapped with the backward phase of one training iteration. We refer to our algorithm as *CPU-GPU data parallel* (CGDP) training.

To obtain the overlap of computation and communication in our algorithm, GPU streams are used. In GPU-CUDA programming, operations in the same stream are executed in a sequential way, while operations in different streams runs in parallel. If there is no stream specified for an operation, the default stream is used. We maintain three streams in our algorithm. The first stream (*default stream*) is used to compute the loss function in the forward phase and the gradients in the backward phases, and is also used to update parameters. The second stream (*D2H stream*) is used to send local gradients to host and then call a callback function on host to accumulate gradients. The third stream (*H2D stream*) is used to broadcast the global gradient back to the GPUs. Note that, using more streams do not make data transfers done in parallel because only one transfer in one direction is allowed at a time.

One iteration in our CGDP training is as follows:

Step P-1: Every GPU reads one minibatch and performs the forward phase.

Step P-2: Every GPU performs the backward phase (being explained soon).

Step P-3: Every GPU performs the update phase to update its learnable parameters.

During the backward phase (Step P-2), gradients accumulations are performed on host, and the accumulated gradients are broadcasted to all GPUs. Hence, at the end of the backward phase, all GPUs have the same accumulated gradients and they update its learnable parameters in parallel. This is different from the original data parallel training in which only the GPU 0 has the accumulated gradients and does the update phase. In other words, there is no server GPU in the CGDP training.

Step P-2 is an extension of the original back-propagation algorithm (Section 2.2) by inserting four steps: Steps B-2-0-1 and B-2-0-2 before Step B-2, Step B-3-1 after Step B-3 and Step B-5-1 after Step B-5. Let Q be a queue to store layers whose backward computation has already done. If a layer is in Q, its gradient is sending to host or accumulating into the global gradient on host. The four steps are as follows (assume that we are on GPU j):

Step B-2-0-1: Synchronize host with respect to the D2H stream.

Step B-2-0-2: For each layer q in Q, if all local gradients $\nabla^j_{w_q} J$, $j = 0, \ldots, (G-1)$ are accumulated into a global gradient then use the H2D stream to broadcast the global gradient to all GPUs and pop q from Q.

Step B-3-1: Use the D2H stream to send local gradients $\nabla^j_{w_k} J$ and $\nabla^j_{b_k} J$ to host, and call a callback function to accumulate the local gradients into the global gradient on host. Push k to Q.

Step B-5-1: For each layer q in Q, if all local gradients $\nabla^j_{w_q} J$, $j = 0, \ldots, (G-1)$ are accumulated into the global gradient then use the H2D stream to broadcast the global gradient to all GPUs and pop q from Q. Repeat this step until Q is empty.

where a global gradient of a layer is a gradient accumulated from all local gradients of that layer from all GPUs. It is worth noting that the synchronization step (Step B-2-0-1) at the beginning of a layer's backward phase is important. It ensures that a local gradient of a layer is sent to host after the computation for the gradient totally finished.

Figure 2 shows the communication pattern between host and GPUs during the backward phase of one iteration. On host, for each GPU, there is a concurrent vector of local gradients produced by layers. Additionally, there is another concurrent vector of global gradients accumulated from the local gradients. GPUs communicate directly with the CPUs to send gradients to the CPUs. Once a layer computed gradients with respect to its learnable parameters, the gradients are sent to host where they are accumulated into the global gradients (Step B-3-1). Gradient accumulation on host is done in parallel using OpenMP. At the same time, the next layer computes other gradients. At the beginning of a layer in the backward phase, we always check the accomplishment of gradient accumulation for the layers in the queue Q and broadcast their global gradients back to all GPUs (Step B-2-0-2). Note that, once all layers finished their computations, the accomplishment of gradient accumulation for the last layer (maybe some last layers) hasn't finished yet. Hence, we need another step at the end of the backward phase to check the accomplishment of the layer(s) and broadcast the remaining global gradient(s) to the GPUs (Step B-5-1).

3.2 A cost model

We design a cost model for the CGDP training and analyze its performance. Without loss of generality, we assume that every GPU trains the same neural network in parallel at the same pace. In other words, the same layers finish its computation at the same time. Hence, it is enough to consider only the training on one GPU. Furthermore, we consider only the backward phase because we do not change the forward and update phases.

Given a layer f^i, let t^i_{bp} be the back-propagation time on GPU of the layer (Step P-2), t^i_a be the time on host for gradient accumulation (Step B-3-1), t^i_{bc} be the time for broadcasting gradient from host to device (Step B-5-1). t_a includes the synchronization time (t_{as}) and accumulation time (t_{aa}), where t_{as} is the time taken for the synchronization between the default stream and the D2H stream, t_{aa} is the time on host for accumulating all gradients of the layer generated by all GPUs (including gradient copy from device to host). In the CGDP training, t_{aa} and t_{bc} are overlapped with the next t_{bp}(s). In addition, t_{aa}(s) of different layers are overlapped because they are handled by different processes in parallel.

Given an l-layer FFN, f_θ, the running time of the backward phase using the CGDP training, T, is computed as follows:

$$\begin{aligned} T^i_{BP} &= \sum_{l}^{i} (t^i_{bp} + t^i_{as}) + t^i_{aa} + t^i_{bc} \\ T &= \max_{i=1,\ldots,l} (T^i_{BP}) \end{aligned} \tag{1}$$

where T^i_{BP} is called the *total time* of a layer i, measuring the time from the beginning of the backward phase to the point where the i-layer's accumulated gradient is available in the GPU. The definition of T^i_{BP} is interesting in the sense that althouth it does not guarantee that, for $i < j$, the layer j will finish before the layer i during the backward phase, it is powerful to analyze performance as we will show later. The runtime of the backward phase in the original data parallel training, T', is computed as follows (in this case, there is no synchronization because we use only one stream, the default stream, or $t_{as} = 0$):

$$T' = \sum_{i=l}^{1} (t^i_{bp} + t^i_{aa} + t^i_{bc}) = T^1_{BP} + \sum_{i=l}^{2} (t^i_{aa} + t^i_{bc})$$

It is worth noting that t_{bc} in T is the time for broadcasting gradients from host to every GPU, while t_{bc} in T' is the time for broadcasting parameters from one GPU to the other GPUs. For one layer, the sizes of gradients and parameters are the same. Hence, we assume that these t_{bc}(s) are the same though they might be different due to different connection topologies among CPUs and GPUs. In addition, t_{aa} in T is performed by CPUs while the one in T' is performed by GPUs.

Let $T_{BP} = \sum^1_{i=l} (t^i_{bp})$. Overhead time in the original data parallel training is defined by

$$T'_O = T' - T_{BP} = \sum_{i=l}^{1} (t^i_{aa} + t^i_{bc})$$

Overhead time in the CGDP training is non-trivial to formalize. However, by simplifying it using an assumption that t_{aa} and t_{bc} of layers $l, l-1, \ldots, 2$ are perfectly overlapped with the next t_{bp}(s), it means $T = T^1_{BP}$, the overhead is defined by

$$T_O = T - T_{BP} = \sum_{i=l}^{1} (t^i_{as}) + t^1_{aa} + t^1_{bc} \tag{2}$$

The CGDP training is faster than the original data parallel training if $T_O < T'_O$. If the layer 1 has a small number of parameter, then t_{aa} in T_O is approximately equal to t_{aa} in T'_O. Hence, $T_O < T'_O$ holds if $\sum^1_{i=l} (t^i_{as}) < \sum^2_{i=l} (t^i_{aa} + t^i_{bc})$. This condition is easy to meet for some practical neural networks such as AlexNet, GoogLeNet-v1, and VGGNet-16.

4 CHUNK_SIZE OPTIMIZATION

In this section we propose a fine-grained optimization for the CGDP training showed in Section 3.

In the naive CGDP training, gradients are sent to host layer-by-layer, which is triggered by a synchronization between the default stream and the D2H stream. In other words, the D2H stream waits

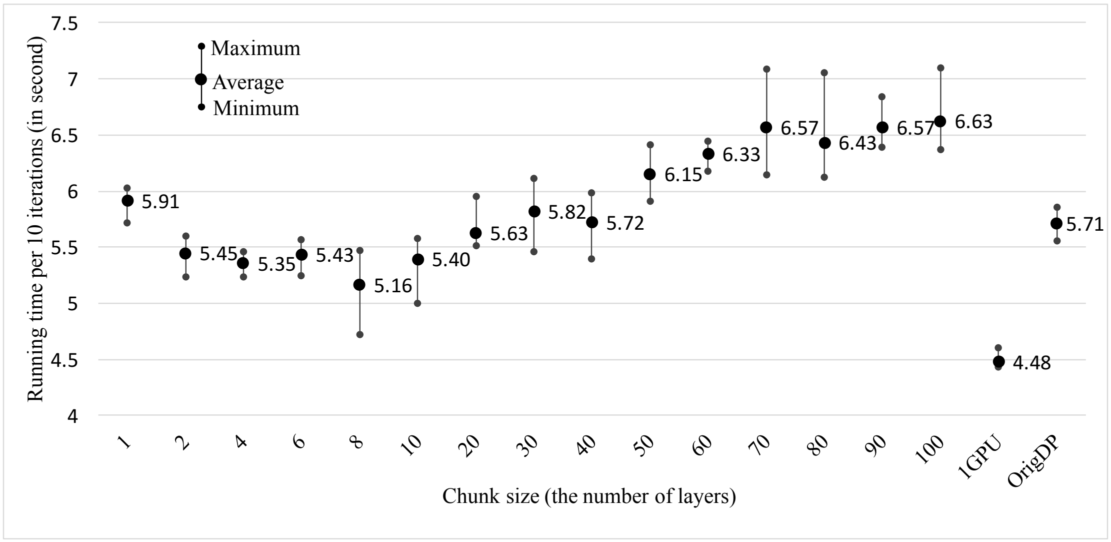


Figure 3: Running time for 10 iterations of training Resnet-152 (minibatch size 16, four GPUs, no validation) when varying the chunk size. "OrigDP" means using the original data parallel training. For each 10 iterations, we take the maximum, minimum and average values.

for the computation of a layer in the default stream finished. For neural networks that have many layers, e.g. Resnet-152 with 152 layers, there are many synchronizations in the CGDP training, which slows down the backward phase. We first extend the naive CGDP training by using chunks to reduce the effect of synchronizations on the performance of the backward phaseand, and then propose a run-time algorithm to automatically find a good setting for the CGDP training with chunks.

4.1 CGDP training by chunks

As stream synchronization slows down the performance of the backward phase, it is better to do synchronization after some layers instead of one layer. In particular, we would like to optimize the part $(\sum_{i=1}^{l}(t_{as}^{i}) + t_{aa}^{i} + t_{bc}^{i}))$ of T in Equation 1.

We call a group of layers whose gradients are sent to host together *a chunk of layers*, and denote it by $\{\}$. Given a neural network, we can use multiple chunks of layers with different sizes for the CGDP training. For example, if a neural network has 6 layers: $f_{\theta_1}^{1}, f_{\theta_2}^{2}, f_{\theta_3}^{3}, f_{\theta_4}^{4}, f_{\theta_5}^{5}, f_{\theta_6}^{6}$, we could use 3 chunks $\{f_{\theta_1}^{1}\}$, $\{f_{\theta_2}^{2}, f_{\theta_3}^{3}\}$, and $\{f_{\theta_4}^{4}, f_{\theta_5}^{5}, f_{\theta_6}^{6}\}$. The backward phase of the CGDP training with chunks is performed as follows: compute gradients for the layers of the chunk $\{f_{\theta_4}^{4}, f_{\theta_5}^{5}, f_{\theta_6}^{6}\}$ using the default stream, synchronize with the D2H stream to send the gradients of these layers to host, continue to compute gradients for the layers of the chunk $\{f_{\theta_2}^{2}, f_{\theta_3}^{3}\}$ using the default stream, synchronize with the D2H stream to send the gradients of these layers to host, continue to compute gradients for the layers of the chunk $\{f_{\theta_1}^{1}\}$, synchronize with the D2H stream to send the gradients of these layers to host, and wait for all gradients are available in GPUs. It is clear that we need only three synchronizations using chunks instead of six synchronizations without chunks.

There exists a tradeoff between the number of synchronizations and the number of layers in a chunk in the CGDP training with chunks. It is obvious that the CGDP training with chunks reduces the number of synchronizations because $\sum_{i=1}^{l}(t_{as}^{i})$ becomes $\sum_{j=1}^{c}(t_{as}^{j})$, where c is the number of chunks. Nevertheless, using chunks potentially produces more overhead since we have postponed gradients accumulations of layers in a chunk until the last layer in the chunk finishes its back-propagation. In other words, $(t_{aa}^{i} + t_{bc}^{i})$ for a layer i becomes the sum of $(t_{aa}^{j} + t_{bc}^{j})$ of all layers j in the chunk to which the layer i belongs. This makes the CGDP training difficult to be optimized.

Proposition 4.1. *Given an l-layer FFN, there are $\sum_{k=1}^{l-1}\binom{l-1}{k}$ ways to group layers by chunks for the CGDP training with chunks.*

Proof. The proof is completed by counting the total number of ways to insert k delimiters, $k = 1, 2, \ldots, (l-1)$, into the spaces between two consecutive characters in the sequence "$f_1 f_2 \ldots f_l$", so that there is no more than one delimiter in a space. □

4.2 Heuristic algorithm for finding chunks

In this section, we propose a run-time algorithm to find a good chunk size so that the time for the backward phase is minimized. In general, our algorithm runs for some first training iterations and tries to determine the best chunk size at which the running time for training is smallest. We provide two heuristic rules to decide how to expand the search space and how to stop the algorithm.

To narrow the search space, the paper considers only the case where chunks have the same size except some last layers of the backward phase. Assume that we train an l-layer FFN using the CGDP training with chunks of the same size k. If (l mod $k = 0$), there are $(\frac{l}{k} - 1)$ chunks with the size k, including layers from l to $(k+1)$, and there are k chunks with the size 1, including the

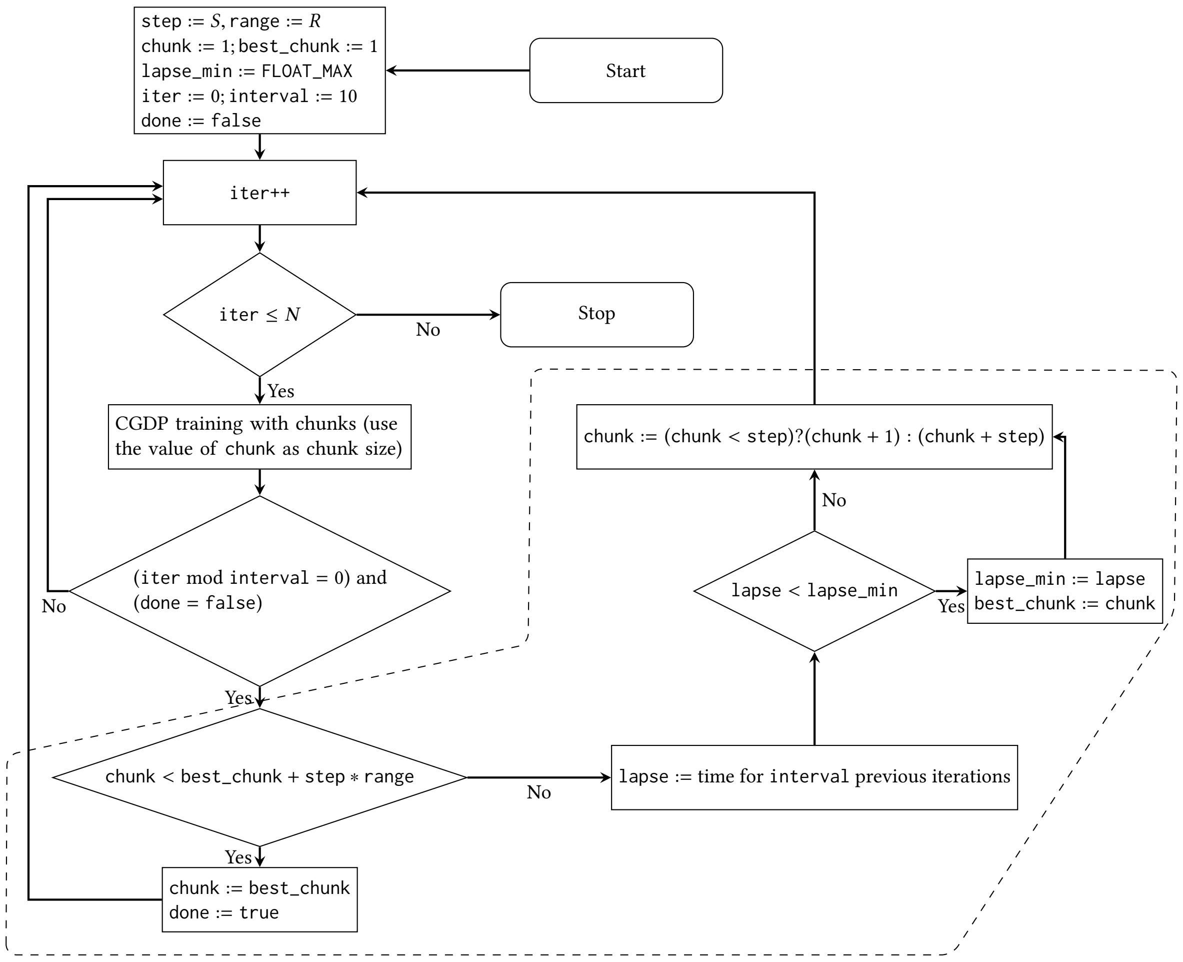

step := S, range := R
chunk := 1; best_chunk := 1
lapse_min := FLOAT_MAX
iter := 0; interval := 10
done := false
Start
iter++
iter ≤ N
No
Stop
Yes
CGDP training with chunks (use the value of chunk as chunk size)
(iter mod interval = 0) and (done = false)
No
Yes
chunk := (chunk < step)?(chunk + 1) : (chunk + step)
No
lapse < lapse_min
Yes
lapse_min := lapse
best_chunk := chunk
chunk < best_chunk + step ∗ range
No
lapse := time for interval previous iterations
Yes
chunk := best_chunk
done := true

Figure 4: Run-time algorithm for finding a good chunk size for the CGDP training with chunks. N is the number of iterations. S and R are inputs.

remaining layers from k to 1. If (l mod $k \neq 0$) there are $\left\lfloor \frac{l}{k} \right\rfloor$ chunks with the size k, including layers from l to $(l - k * \left\lfloor \frac{l}{k} \right\rfloor + 1)$, and there are $(l - k * \left\lfloor \frac{l}{k} \right\rfloor)$ chunks with the size 1, including the remaining layers from $(l - k * \left\lfloor \frac{l}{k} \right\rfloor)$ to 1. Some last layers have a chunk size of 1 because we would like to reduce the effect of t^i_{aa} and t^i_{bc} on the overhead of training.

Figure 3 shows the running time for training Resnet-152 when varying the chunk size from 1 to 100. It also shows the running times of single GPU training and the original data parallel training, which helps understand the overhead of the CGDP training. In this case, the naive CGDP training (with chunks of size 1) is slower than the original data parallel training. It is because Resnet has many small layers (the number of parameters is small). Hence, the overhead of synchronizations is high, which results in that the total overhead is high. When we slowly increase the chunk size, we start to get better results (chunk sizes 2, 4, 6, 8). However, using a big chunk size is not good due to the overhead of gradient accumulation.

Figure 4 shows our algorithm for finding a good chunk size for the CGDP training with chunks during training. Assume that we train a neural network with N iterations. There are two user-defined parameters in the algorithm: `step` and `range`. The parameter `step` is used as a heuristic parameter to define how to expand the search space of chunk size and how to stop the algorithm. The parameter `range` is to determine how to stop the algorithm and it is used together with the parameter `step`. A variable `chunk` is used to store the chunk size for the current iteration, and is the variable we want to optimize. It is updated during the algorithm and is set to the value of variable `best_chunk` once the algorithm stopped. The variable `best_chunk` holds the chunk size that results in the minimum running time that is stored in a variable `lapse_min`. A variable `lapse` is the running time of the last `interval` iterations.

Our algorithm (the part inside the dashed line in Figure 4) runs as follows. It first initilizes the chunk size (`chunk`) and the best chunk size (`best_chunk`) to 1, and the minimum running time (`lapse_min`) to the maximum value (mathematically, $+\infty$). After each `interval` iterations, the algorithm is triggered. Users can change the value of `interval`. However, we use a fixed value of 10 for it in this paper. Then, the algorithm measures the running time for the last `interval` iterations, and stores it in the variable `lapse`. If `lapse` <

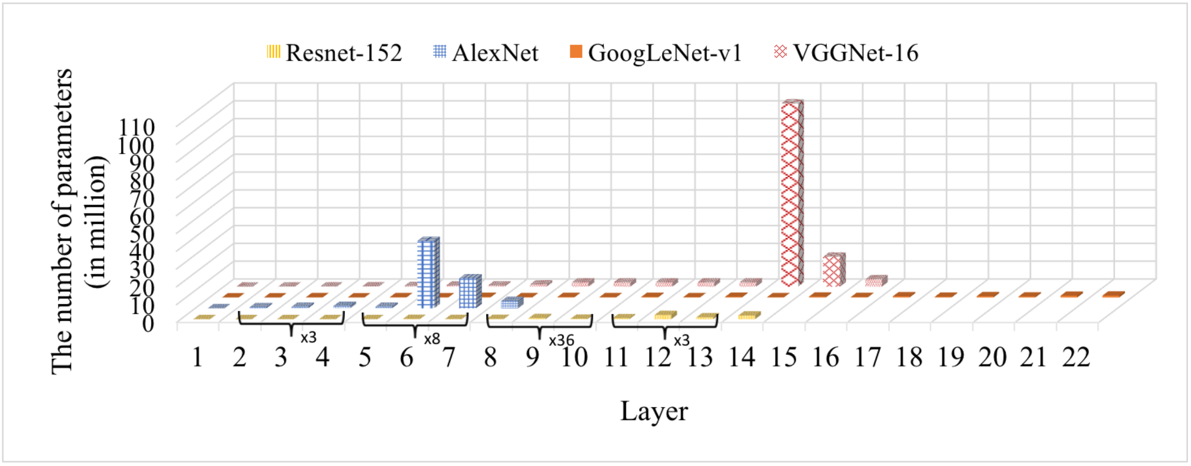


Figure 5: The number of parameters per layer in AlexNet, GoogLeNet-v1, VGGNet-16 and Resnet-152. Strings "x3", "x8", "x36", "x3" are used for Resnet-152 only. They show how many blocks of layers, e.g. there are three blocks of layers 2, 3, 4.

Table 1: Neural networks and experimental settings.

Network	*Layers*	*Parameters (million)*	*Minibatch size per GPU*
AlexNet	8	60	256
GoogLeNet-v1	22	7	64
VGGNet-16	16	138	32
Resnet-152	152	49	12

`lapse_min`, it updates the values of `best_chunk` and `lapse_min` to the current values. Next, it expands the search space of the chunk size with a heuristic rule, chunk := (chunk < step)?(chunk + 1) : (chunk+step). The rule says that, at the beginning of the algorithm, where chunk < step, we slowly increase the value of the chunk size by 1. Otherwise we increase the chunk size by step. This rule is flexible to adjust the search space of the algorithm. If step is large and close to the number of layers, we aggresively scan most of the values for the chunk size. If it is small, we make big jumps for the chunk size and ignore some values to finish early. Another rule is to determine when to stop the algorithm, that is chunk < best_chunk + step ∗ range. Intuitively, it says that once we found a best_chunk, we runs the algorithm another range times. If there is no a better chunk size, the algorithm is stopped. It is note that if best_chunk < step, the algorithm may run another (step − best_chunk + range) times before deciding to stop.

5 EXPERIMENTAL RESULTS

5.1 Configurations

Experiments were run on an IBM POWER8 NUMA-based machine [1]. It has two 4GHz 10-core POWER8 processors, eight SMTs per core and 256 MB RAM per processor, four NVIDIA Tesla P100 GPUs (each has 16 GB memory), NVLinks among GPUs and CPUs (one 80 GB/s duplex link between GPUs 0 and 1, one 80 GB/s duplex link between GPUs 2 and 3, two 80 GB/s duplex links from CPU 0 to GPUs 0,1, and two 80 GB/s duplex links from CPU 1 to GPUs 2,3). CUDA Toolkit v8.0.44 and cuDNN 5.1.5 were installed on the machine. cuDNN is a state-of-the-art library for primitives used in deep neural networks, which is developed by NVIDIA developers and highly optimized for GPUs.

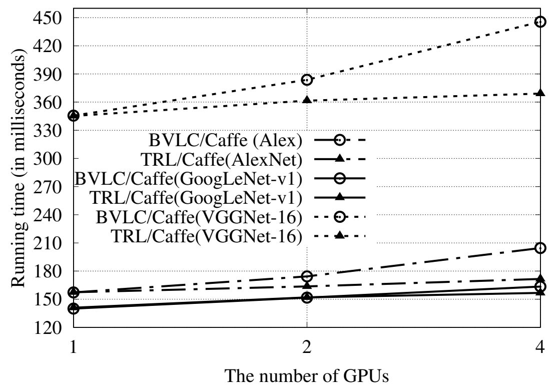
We used four neural networks that are widely used in computer vision: AlexNet[1] [17], GoogLeNet-v1[2] (Inception-v1) [23], VGGNet-16[3] (model D) [21], and Resnet-152[4][11]. Table 1 shows the basic information for these networks and the size of the minibatch (number of images processed by *one* GPU in one training iteration) used for training them. Distributions of parameters in layers of these networks are shown in Figure 5. A training dataset for experiments is a subset of ImageNet ILSVRC2012 [19] that contains 1.2 million images classified into 1000 categories.

We implemented our optimization in the BVLC/Caffe deep learning framework [15] developed by UC Berkeley researchers. The vanilla BVLC/Caffe v1.0.0-rc3 (hereafter, BVLC/Caffe) uses the standard data parallel training with a tree pattern for communication among GPUs. We refer to our optimization in BVLC/Caffe as TRL/Caffe. To obtain exact results, for each training session, we ran the program ten times and calculated the average execution time. Each training session comprised 1000 training iterations. The running time for an iteration was averaged on the basis of 1000 iterations.

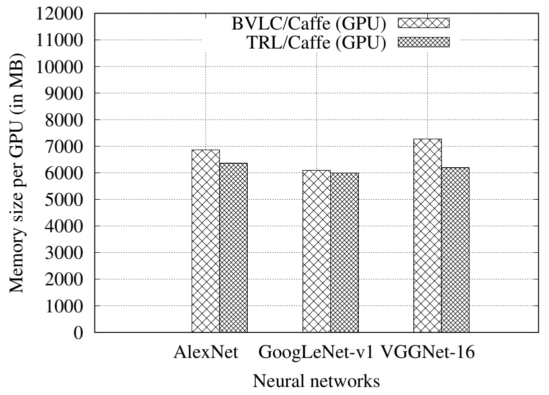
5.2 Results

5.2.1 Execution time for training. Figure 6a shows the execution times for one training iteration with one, two, and four GPUs for AlexNet, GoogLeNet-v1, and VGGNet-16. The results indicated that TRL/Caffe was more scalable than BVLC/Caffe. When the number

[1]AlexNet's network definition: https://github.com/BVLC/caffe/tree/master/models/bvlc_alexnet
[2]GoogLeNet-v1's network definition: https://github.com/BVLC/caffe/tree/master/models/bvlc_googlenet
[3]VGGNet-16's network definition: https://gist.github.com/ksimonyan/211839e770f7b538e2d8
[4]Resnet-152's network definition: https://github.com/KaimingHe/deep-residual-networks



(a) Execution time for one training iteration.



(b) Memory consumption in GPUs.

Figure 6: Execution time and memory consumption.

of GPUs was one, the execution times in all frameworks were almost the same for each network. When the number of GPUs was four, TRL/Caffe was the fastest in all networks—in particular, it was 1.19, 1.04, and 1.21 times faster than BVLC/Caffe for AlexNet, GoogLeNet-v1, and VGGNet-16, respectively. This showed that TRL/Caffe consistently had a high efficiency ($\geq$ 90%). Our approach was the least effective for GoogLeNet-16 and the most effective for VGGNet-16, respectively. In fact, the effectiveness of our approach depends on the number of parameters for the network: it makes the training of networks with more parameters faster because it distributes the computation for collecting and accumulating gradients, the number of which is the same as the number of parameters for any minibatch size. The numbers of parameters in GoogLeNet-v1 and VGGNet-16 are the least and the most, respectively, so we obtained corresponding results.

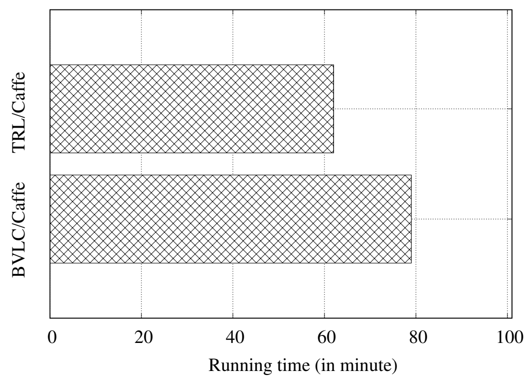
5.2.2 Memory consumption. Figure 6b shows the maximum sizes of the memories allocated on GPUs during training. Note that the allocated memory size on the GPUs depends on the size of the minibatch used for training, not the number of iterations. The maximum size per GPU needed by TRL/Caffe was smaller than the one needed by BVLC/Caffe. This is because BVLC/Caffe allocates memory on the GPUs to collect and accumulate gradients from different GPUs while TRL/Caffe does not need such memory since the gradients are collected and accumulated on host. Accordingly, TRL/Caffe consumed more memory on host. This is not a big problem because memory on host is generally cheaper and easier to increase than that on GPUs. As a result, we were able to train VGGNet-16 with 103 images per minibatch per GPU instead of 94, increasing the chance to adjust the hyperparameters for the solver algorithms.

5.2.3 Long-term runs. We conducted experiments using long-term runs to see how our approach is effective for long-term runs. The experiments were to train AlexNet to achieve 50% accuracy using 4 GPUs. TRL/Caffe took 62 minutes while BVLC/Caffe took 79 minutes (Fig. 7a). Both versions reached 50% accuracy at around iteration 20, 000 (21, 000 iterations in total), and had the same convergence curve (Fig. 7b). Note that this training included a testing phase, in which a testing iteration was simply a forward computation using validation data (50, 000 images) to verify network accuracy. After 1000 training iterations, there was one test comprising 1000 testing iterations. Hence, there were 21 tests in total, and each test took about 14.7 seconds.

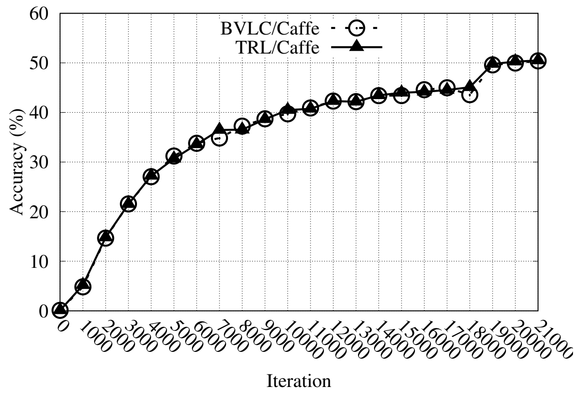
5.2.4 Communication overhead. Reducing communication overhead is our objective, and Table 2 shows the execution time by phase for AlexNet when using 4 GPUs. We see that, for BVLC/Caffe, the broadcast at the beginning took 20 ms and that the collection and accumulation of gradients at the end of the backward phase took 23 ms. For TRL/Caffe, these computations were hidden behind the backward phase, and the communication overhead was for only the ending layer of the backward phase. These computations took only 21.8μs in AlexNet. However, the time for backward propagation in TRL/Caffe was longer than the one in BVLC/Caffe. This is reasonable because TRL/Caffe needs to do some works to invoke data copy functions and callback functions between two consecutive layers during the backward phase. The communication overheads of TRL/Caffe for GoogLeNet-v1 and VGGNet-16 were the same as the one in AlexNet ($\approx$ 21.8μs). Hence we do not show them here.

5.2.5 Time for accumulation on CPU. Table 3 shows the time for accumulation on the CPUs for each layers during the backward phase and the communication time between the GPUs and the CPUs for TRL/Caffe with AlexNet when using 4 GPUs. The communication and accumulation overlapped the backward phase. Note that in the backward phase, processing is from the top layer (layer 8) to the bottom layer (layer 1). It is clear that the accumulation time was much shorter than the time for the backward phase. Furthermore, because the ending layer in the backward phase, layer 1, had a small number of parameters, the overhead for sending the gradients of this layer to the GPUs was very small ($\approx$ 7μs).

5.2.6 Effectiveness of the run-time algorithm. As Resnet-1001 is too large to fit in our GPU memory, we used Resnet-152 instead for measuring the effectiveness of the run-time algorithm. We trained Resnet-152 for 1000 iterations using 4 GPUs with a real setting in which the validation phase was included. The maximum minibatch



(a) Time to achieve 50% accuracy in AlexNet training.



(b) Accuracy per iteration in AlexNet training.

Figure 7: Long-term run for AlexNet training (21, 000 iterations).

Table 2: Execution time for phases in one iteration for AlexNet with four GPUs (in ms).

	broadcast	*forward*	*backward*	*broadcast the remaining gradients from the CPUs*	*grad-acc*	*update-param*	*total time*
BVLC/Caffe	20	50.6	104	N/A	23	5.1	202.7
TRL/Caffe	N/A	51.0	111	21.8μs	N/A	5.1	167.1

Table 3: Communication time between GPUs and CPUs, and accumulation time on CPUs.

	Layer 8	*Layer 7*	*Layer 6*	*Layer 5*	*Layer 4*	*Layer 3*	*Layer 2*	*Layer 1*
no. of parameters (million)	4	16.8	37.8	0.4	0.7	0.9	0.3	0.03
GPU-to-CPU copy (ms)	0.760	3.156	12.146	0.133	0.174	0.275	0.066	0.014
accumulation time on CPUs (ms)	1.263	4.441	13.074	0.285	0.465	0.578	0.155	0.018
CPU-to-GPU copy (ms)	1.786	2.568	5.538	0.064	0.095	0.123	0.041	0.007

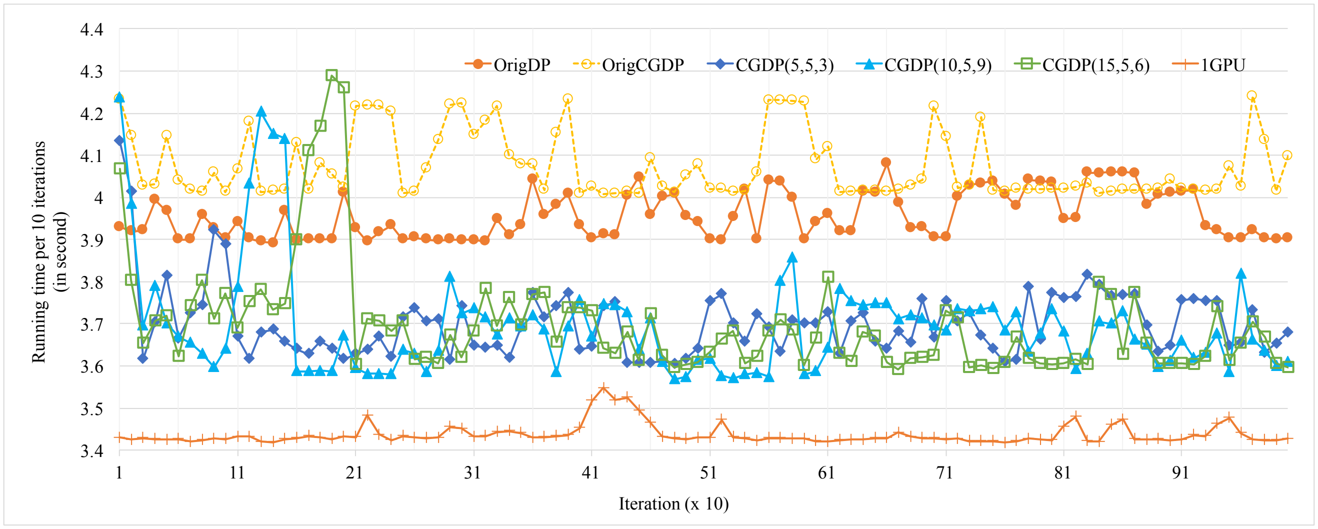
size we were able to run for Resnet-152 was 12 (16 without the validation phase). We set the value of `interval` to 10 to make the elapsed time stable. Effectiveness of each parameter in the run-time algorithm was examined.

We first fixed the value of `range` to 5, and varied the value of `step`. Figure 8a shows the result. First, for the chunk size of 1 (OrigCGDP), the CGDP training was still slower than the original data parallel training. Second, when we changed the value of `step`, different values might lead to a different best chunk size. However, once the run-time algorithm finished, the trainings with different best chunk size looked working similarly and ran faster than the original data parallel training. Let's analyze in detail the configuration "`step` = 10, `range` = 5". The CGDP training had the best performance at iteration 90 where the chunk size had a value of 9. After that, it could not get a better running time. Hence, the run-time algorithm stopped at iteration 150 (ran 6 times after the iteration 90), and used the chunk size of 9 for the later iterations. Finally, we also included a result of single GPU training. Although we still saw an overhead for the CGDP with chunks, with a simple heuristics, it was much better than the original data parallel.

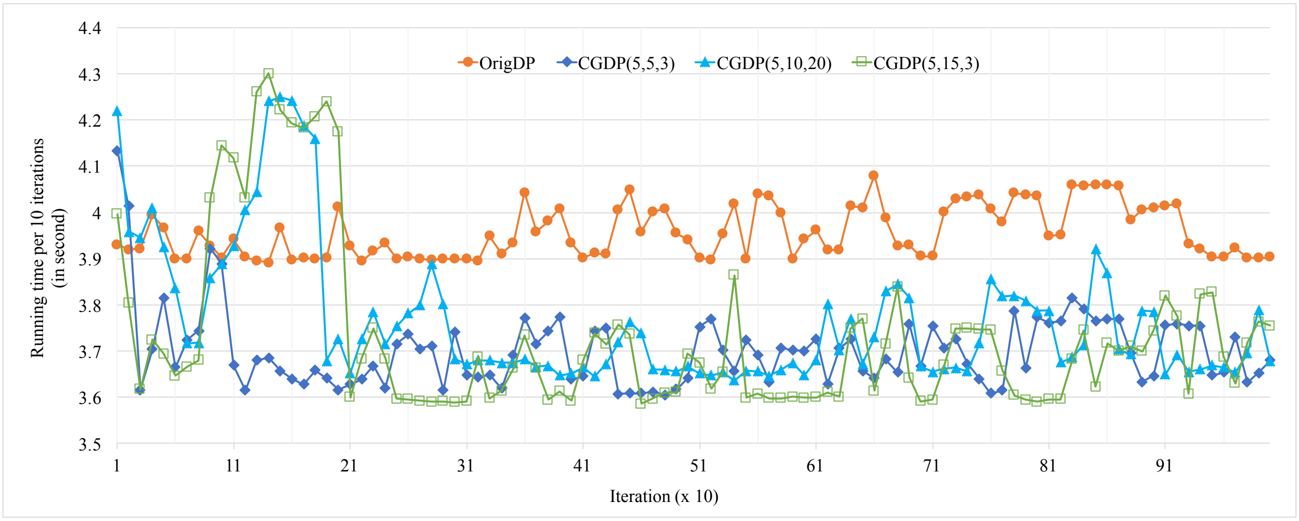
In Figure 8b, we extended the search space for a fixed value of `step` by increasing the value of `range`. We could find a larger value for the chunk size, that was 20, when `range` was 10. Nevertheless, the training with the chunk size of 20 had the same performance as the ones with other best chunk sizes 3, 6, or 9. In all trainings, the run-time algorithm finished in at most 21 iterations, which showed that the overhead of the run-time was small because the whole training often had hundreds of thousands of iterations. Overall, the CGDP training was about 1.07 times faster than the original data parallel training for Resnet-152. For AlexNet, VGGNet-16 and GoogLeNet-v1, the run-time algorithm could not find a good chunk to produce a better result. These neural networks have a small number of layers, hence the effect of synchronization is small.

6 RELATED WORK

There are several ways to accelerate deep learning: data parallelism, model parallelism, and pipeline parallelism. Data parallelism is implemented in many frameworks such as Google's TensorFlow [5], Torch [8], and Microsoft's CNTK [24]. It is mainly used for deep convolutional neural networks. Model parallelism has been used



(a) `step` ∈ {5, 10, 15}, `range` = 5.



(b) `step` = 5, `range` ∈ {5, 10, 15}.

Figure 8: Effectiveness of the parameters in the run-time algorithm for Resnet-152, minibatch size 12 per GPU. CGDP(`step`, `range`, `best_chunk`) where `best_chunk` is the best chunk found by the algorithm. "OrigDP" refers to the original data parallel training. "OrigCGDP" refers to the naive CGDP training with the chunk size of 1.

for large-scale unsupervised learning [18]. Many distributed frameworks, such as MXNet [6], Mariana [27], COTS HPC system [7], and DistBelief [9], etc., support both data parallelism and model parallelism so that users can choose either one to use. Krizhevsky proposed a hybrid parallelism [16], in which model parallelism is applied to layers with a large number of learnable parameters (e.g., fully connected layers), and data parallelism is applied to the ones with a small number of learnable parameters (e.g., convolutional layers). This hybrid parallelism scales better than model and data parallelism when applied to modern convolutional neural networks. In pipeline parallelism, each layer of a neural network is executed on a different GPU and communicates its activations to the next GPU [22]. Pipeline optimization is used in Mariana [27]: a three-stage of pipeline for training, consisting of data reading, data processing and neural network training is used for training. Our mechanism should also be effective for hybrid parallelism and pipeline parallelism because both require collection and accumulation of gradients on different GPUs.

A closer approach to ours is Poseidon [25], a distributed deep learning framework, in which there is overlap between backward computation and communication among distributed machines. Because communication overhead is very high in a distributed environment, it is difficult to hide communication overhead behind the backward phase. It can be done with our approach because our target is training on a single machine coupled with multiple GPUs.

Another approach to accelerating deep neural network training is to parallelize the solver algorithm, such as the SGD algorithm. BVLC/Caffe implements the synchronous SGD algorithm so that parameters are updated when *all* gradients have been collected from *all* GPUs [15]. Asynchronous SGD is a parallelized variant of SGD, in which the parameters are updated after a certain number of gradients have been collected from a certain number of GPUs. It is very effective in a distributed environment where different machines run at different speeds [14, 26], but it changes the learning accuracy and sometimes it is difficult to exactly reproduce the result.

7 CONCLUSION AND FUTURE WORK

Deep learning is an emerging tool for signal processing and is mainly speeded up by accelerators such as GPUs. This leads to a consequence that powerful CPUs are redundant in deep learning. Some frameworks, such as TensorFlow or MXNet, have been utilizing CPUs for gradient accumulation, but their purpose is to make the frameworks flexible instead of improving performance. In this paper, we have shown for the first time to our knowledge that free CPUs on host certainly help accelerate GPU-based data parallel training of deep neural networks in a single machine. We designed an approach, named CGDP training, to utilize CPUs for collecting, accumulating, and broadcasting gradients produced during the backward phase. The key idea is that those operations can be performed in parallel with the backward phase, resulting in reduction in time for one training iteration. We then provided a cost model for data parallel training of neural networks. We showed the power of the model by using it to determine a bottleneck in training very deep neural networks. Finally, we proposed a run-time algorithm with a simple heuristics to optimize one of the deepest neural network, Resnet-152. Experimental results for the state-of-the-art deep neural networks showed that the communication overhead in the CGDP training was very low (microsecond scale). The CGDP training consistently achieved more than 90% weak scaling efficiency for three networks Alexnet, GoogLeNet-v1 and VGGNet-16, and it significantly reduced the training time for long-term runs with large datasets (e.g., ImageNet). For very deep neural networks such as Resnet-152, the CGDP training was 1.07 times faster than the original data parallel training. In the future, we will extend the CGDP training to support recurrent neural networks (RNN) that consist of cyclic links among layers. For RNNs, an additional mechanism is needed to determine which layer gradients are kept in GPUs for the cyclic links, and when is a good timing to send them to host. Additionally, it is open to further investigate what kind of computation is suitable for being offloaded to host.

REFERENCES

[1] 2016. IBM Power System S822LC for High Performance Computing. (Oct. 2016). http://www-03.ibm.com/systems/power/hardware/s822lc-hpc/.

[2] 2016. Torch. (Oct. 2016). http://torch.ch/.

[3] 2017. NVIDIA NCCL. (2017). https://developer.nvidia.com/nccl.

[4] 2017. Torch. (2017). https://luna16.grand-challenge.org.

[5] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. (2015). http://tensorflow.org/ Software available from tensorflow.org.

[6] 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).

[7] Adam Coates, Brody Huval, Tao Wang, David Wu, Bryan Catanzaro, and Ng Andrew. 2013. Deep Learning with COTS HPC Systems. In *International Conference on Machine Learning*, Vol. 28. JMLR Workshop and Conference Proceedings, 1337–1345.

[8] Ronan Collobert, Koray Kavukcuoglu, and Clément Farabet. 2011. Torch7: A Matlab-like Environment for Machine Learning. In *BigLearn, NIPS Workshop*.

[9] Jeffrey Dean, Greg S. Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Quoc V. Le, Mark Z. Mao, MarcĄfAurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, and Andrew Y. Ng. 2012. Large Scale Distributed Deep Networks. In *International Conference on Neural Information Processing Systems*. 1232–1240.

[10] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. 2016. *Deep Learning*. MIT Press. http://www.deeplearningbook.org.

[11] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Deep Residual Learning for Image Recognition. *CoRR* abs/1512.03385 (2015). http://arxiv.org/abs/1512.03385

[12] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification. *CoRR* abs/1502.01852 (2015).

[13] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. *Identity Mappings in Deep Residual Networks*. Springer International Publishing, 630–645.

[14] Qirong Ho, James Cipar, Henggang Cui, Seunghak Lee, Jin Kyu Kim, Phillip B. Gibbons, Garth A Gibson, Greg Ganger, and Eric P Xing. 2013. More Effective Distributed ML via a Stale Synchronous Parallel Parameter Server. In *International Conference on Neural Information Processing Systems*. 1223–1231.

[15] 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. *arXiv preprint arXiv:1408.5093* (2014).

[16] Alex Krizhevsky. 2014. One Weird Trick for Parallelizing Convolutional Neural Networks. *arXiv preprint arXiv:1404.5997v2* (2014).

[17] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *International Conference on Neural Information Processing Systems*. 1097–1105.

[18] Quoc Le, Marc'Aurelio Ranzato, Rajat Monga, Matthieu Devin, Kai Chen, Greg Corrado, Jeff Dean, and Andrew Ng. 2012. Building High-Level Features Using Large Scale Unsupervised Learning. In *International Conference in Machine Learning*.

[19] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. 2015. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision* 115, 3 (2015), 211–252. https://doi.org/10.1007/s11263-015-0816-y

[20] George Saon, Gakuto Kurata, Tom Sercu, Kartik Audhkhasi, Samuel Thomas, Dimitrios Dimitriadis, Xiaodong Cui, Bhuvana Ramabhadran, Michael Picheny, Lynn-Li Lim, Bergul Roomi, and Phil Hall. 2017. English Conversational Telephone Speech Recognition by Humans and Machines. *CoRR* abs/1703.02136 (2017).

[21] Karen Simonyan and Andrew Zisserman. 2014. Very Deep Convolutional Networks for Large-Scale Image Recognition. *arXiv preprint arXiv:1409.1556* (2014).

[22] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. 2014. Sequence to Sequence Learning with Neural Networks. In *International Conference on Neural Information Processing Systems*. 3104–3112.

[23] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott E. Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. 2015. Going Deeper with Convolutions. In *IEEE Conference on Computer Vision and Pattern Recognition*. 1–9.

[24] Dong Yu, Adam Eversole, Mike Seltzer, Kaisheng Yao, Oleksii Kuchaiev, Yu Zhang, Frank Seide, Zhiheng Huang, Brian Guenter, Huaming Wang, Jasha Droppo, Geoffrey Zweig, Chris Rossbach, Jie Gao, Andreas Stolcke, Jon Currey, Malcolm Slaney, Guoguo Chen, Amit Agarwal, Chris Basoglu, Marko Padmilac, Alexey Kamenev, Vladimir Ivanov, Scott Cypher, Hari Parthasarathi, Bhaskar Mitra, Baolin Peng, and Xuedong Huang. 2014. *An Introduction to Computational Networks and the Computational Network Toolkit*. Technical Report.

[25] Hao Zhang, Zhiting Hu, Jinliang Wei, Pengtao Xie, Gunhee Kim, Qirong Ho, and Eric Xing. 2015. Poseidon: A System Architecture for Efficient GPU-based Deep Learning on Multiple Machines. *arXiv preprint arXiv:1512.06216* (2015).

[26] Wei Zhang, Suyog Gupta, Xiangru Lian, and Ji Liu. 2016. Staleness-aware async-SGD for Distributed Deep Learning. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence (IJCAI'16)*. AAAI Press, 2350–2356.

[27] Yongqiang Zou, Xing Jin, Yi Li, Zhimao Guo, Eryu Wang, and Bin Xiao. 2014. Mariana: Tencent Deep Learning Platform and Its Applications. *Proceedings of VLDB Endow.* 7, 13 (Aug. 2014), 1772–1777.