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, training the neural network, often requires the processing of a large amount of data and can takes days to complete. Data parallelism is widely used for training deep neural networks on multiple GPUs in a single machine thanks to its simplicity. However, its scalability is bound by the number of data transfers, mainly for exchanging and accumulating gradients among the GPUs. In this paper, we present a novel approach to data parallel training called *CPU-GPU data parallel (CGDP) training* that utilizes free CPU time on the host to speed up the training in the GPUs. We also present a cost model for analyzing and comparing the performances of both the typical data parallel training and the CPU-GPU data parallel training. Using the cost model, we formally show why our approach is better than the typical one and clarify the remaining issues. Finally, we explain how we optimized CPU-GPU data parallel training by introducing chunks of layers and present a runtime algorithm that automatically finds a good configuration for the training. The algorithm is effective for very deep neural networks, which are the current trend in deep learning. Experimental results showed that we achieved speedups of 1.21, 1.04, 1.21 and 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.

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 *ICPE '18: ACM/SPEC International Conference on Performance Engineering, April 9–13, 2018, Berlin, Germany*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3184407.3184424

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. 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 a top-5 test error rate of 4.94% [12]. Various the deep neural networks for image recognition have been used for detecting pulmonary nodules in the analysis of lung cancer [4]. Long short-term memory (LSTM) networks have reached a major milestone: a 5.5% word error rate in conversational speech recognition [21].

A deep neural network is a combination of many layers (the deepest network up to date is the 1001-layer ResNet network [13]) and is trained using a large dataset. Training a neural network is mainly based on matrix multiplications and is therefore often accelerated by using GPUs. To fully utilize multiple GPUs for training, data parallelism is often used because 1) it is simple to adapt and extend existing single-GPU training to multiple-GPUs training and 2) it fully utilizes the GPU-aware optimized training in a single GPU. In data parallel training, the same neural network is used for each GPU, but the training is done using different inputs for each GPU. Once the GPUs have finished the forward and backward phases of each training iteration, a *server* GPU accumulates the partial gradients from the other GPUs and updates the learnable parameters. The server GPU then broadcasts the updated parameters 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 broadcasting of gradients. The greater the number of GPUs that are used, the greater the amount of data that are exchanged among GPUs. A simple yet effective solution is using a tree layout of GPUs so that some communications are done in parallel; this is the approach used in the BVLC/Caffe [15] and Torch [2] deep learning frameworks. Another approach is using topology-aware communication libraries such as the NVIDIA Collective Communications Library (NCCL) [3]. The TensorFlow framework [5] does both gradient accumulation and parameter update synchronously on CPUs on the host at the end of the backward phase. This approach provides the flexibility needed for distributed training but is slow for training on a single machine. The MXNet [6] deep learning framework provides a mix of the above approaches:

gradient accumulation is done on the CPUs while parameter updating is done on the GPUs.

In this paper, we first present a novel approach to data parallel training called *CPU-GPU data parallel (CGDP) training* that utilizes free CPU time on the host to speed up the training of deep neural networks on the GPUs. In this approach, gradients are collected and accumulated on the host layer-by-layer during the backward phase. Once a partial gradient for a layer is available in a GPU, it is sent to the host. Gradient accumulation is done by the CPUs while the GPU moves on to computing the partial gradients for the other layers. The accumulated gradients on the host are then sent back to the GPU for updating of the learnable parameters. This approach is particularly effective for convolutional neural networks, which are widely used in image processing. Such 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 a substantial amount of time. Furthermore, since collecting and accumulating the gradients of the starting layers take less time, they are completed immediately after the backward phase with a very low overhead.

Next, we present a cost model for analyzing the performance of data parallel training on multiple GPUs. The model takes into account not only the costs for computation and communication, but also the cost for synchronization among GPU streams, which is important for GPU applications. Using this model, we show a condition under which the CGDP training is better than the typical training. To the best of our knowledge, this is the first time a cost model for data parallel training on multiple GPUs has been 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 for a layer is small and roughly equals the number for the other layers. For such networks, the cost model shows that the synchronization among GPU streams is a bottleneck and slows the training down. In such cases, the layers are grouped into chunks, and synchronizations are done for each chunk. We present a runtime algorithm for automatically determining the synchronization points so that the running time is optimized.

We implemented these ideas in the BVLC/Caffe [15] deep learning framework, which is widely used in deep learning communities. Experiments were done with the ImageNet dataset [20] on an IBM POWER8 machine coupled with four NVIDIA Tesla P100 GPUs [1]. Speedups of 1.21, 1.04, and 1.21 were achieved for three neural networks: AlexNet [17], GoogLeNet-v1 [24], and 16-layer VGGNet (model D) (VGGNet-16 hereafter) [22], respectively, corresponding to more than 90% weak scaling efficiency for all three networks. For a very deep neural network, the 152-layer ResNet network [11] (ResNet-152 hereafter), while the naive CGDP training was slower than the typical data parallel training, using chunks and the runtime algorithm made it 1.07 times faster than the typical data parallel training.

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. Our cost model is also presented in the section. Section 4 presents a variant of the CGDP training that uses chunks of layers in training and a runtime algorithm that automatically optimizes the CGDP training with chunks for very deep neural networks. Section 5 presents the experiment results for a real dataset, ImageNet [20]. Section 6 discusses related work. Section 7 summarizes the key points and mentions future work.

2 DATA PARALLELISM FOR DEEP LEARNING

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

2.1 Training Deep Neural Networks

We first give an overview of training deep neural networks using *feed-forward neural networks* (FFNs), which 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 supervised training, otherwise, unsupervised training. In this paper, we focus on supervised training. 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

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

Feed-forward neural networks are represented 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 the set of the layer parameters $\{\theta_1, \theta_2, \ldots, \theta_l\}$.

A layer k is often defined by an activation function to make the neural network nonlinear. Let y_{k-1} be an input vector of the layer k (the output of the previous layer); 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 matrix w_k is $m \times n$. Vector b_k is the *bias vector* of the layer k and has the size of n. In summary, a layer is parameterized by using 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 such a layer can be applied to other kinds of layers, such as convolutional layers.

Training FFNs is almost always based on using gradients with respect to learnable parameters to decrease the loss function. In general, the training 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 computes a scalar value J_θ. The backward phase computes the gradients of the loss function with respect to the learnable parameters, which are $\nabla_{\theta_1} J, \nabla_{\theta_2} J, \ldots, \nabla_{\theta_l} J$. Actually, we need to compute two gradients $\nabla_{w_k} J$ and $\nabla_{b_k} J$ for a layer k. The update phase updates the 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 using a large training dataset, a minibatch stochastic gradient decent (SGD) algorithm is used. The training is performed iteratively, in which, for each iteration, a minibatch (subset) of examples extracted from the training dataset is used as input.

2.2 Back-propagation Algorithm Used to Compute Gradients

Next, 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 composite functions by propagating the derivative information from the loss function back through the composite functions. The back-propagation algorithm has five steps:

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

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

Step B-2: Compute the gradient of the activation for 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 of 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 B-2 to B-4 until l reaches 1.

The $\left(\frac{\partial y}{\partial x}\right)$ denotes the $m \times n$ Jacobian matrix of a function g for $y = g(x)$, m is the size of vector y and n is the size of 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 use the same neural network and train the network with different minibatches. Each iteration of data parallel training comprises five steps:

Step T-1: Each GPU reads one minibatch and performs the forward phase.

Step T-2: Each GPU performs the back-propagation 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 GPU j.

Step T-3: GPU 0 collects the gradients from the other GPUs and computes their mean value:

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

Step T-4: GPU 0 updates its learnable parameters by using the computed gradients.

Step T-5: GPU 0 broadcasts the values of its learnable parameters to the other GPUs. The other GPUs set the values of their learnable parameters to those values.

In this training, GPU 0 plays the role of a parameter server, collecting gradients and updating the learnable parameters. The backward phase consists of Steps T-2, T-3, and T-5. The training actually begins with Step T-5 to ensure that the neural networks are trained using the same parameters.

3 CPU-GPU DATA PARALLEL TRAINING

The drawback of the typical data parallel training lies in the communication steps (Steps T-3 and T-5 (Section 2.3)), leading to a poor scalability. The two steps depend on the communication pattern among GPUs. Our goal for optimizing data parallel training is to reduce as much as possible the effect of the communication steps on the running time of one training iteration.

3.1 Algorithm

Our algorithm is based on two observations. The first 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 the host. We thus make Steps T-3 (gradient accumulation) and T-5 (parameter broadcast) overlapped with Step T-2 (back-propagation). We refer to our algorithm as *CPU-GPU data parallel (CGDP) training*.

To attain overlap of computation and communication in our algorithm, GPU streams are used. In GPU-CUDA programming, operations in the same stream are executed sequentially, while operations in different streams are executed in parallel. If a stream is not specified specified for an operation, the default stream is used. Three streams are maintained in our algorithm. The first stream, the *default stream*, is used to compute the loss function in the forward phase and the gradients in the backward phases; it is also used to update the parameters. The second stream, the *D2H stream*, is used to send local gradients to the host and then call a callback function on the host to accumulate the gradients. The third stream, *H2D stream*, is used to broadcast the global gradients back to the GPUs. Note that, even if more streams are used, the data transfers are not done in parallel because only one transfer in one direction is allowed at a time.

Each iteration in our CGDP training consists of three steps:

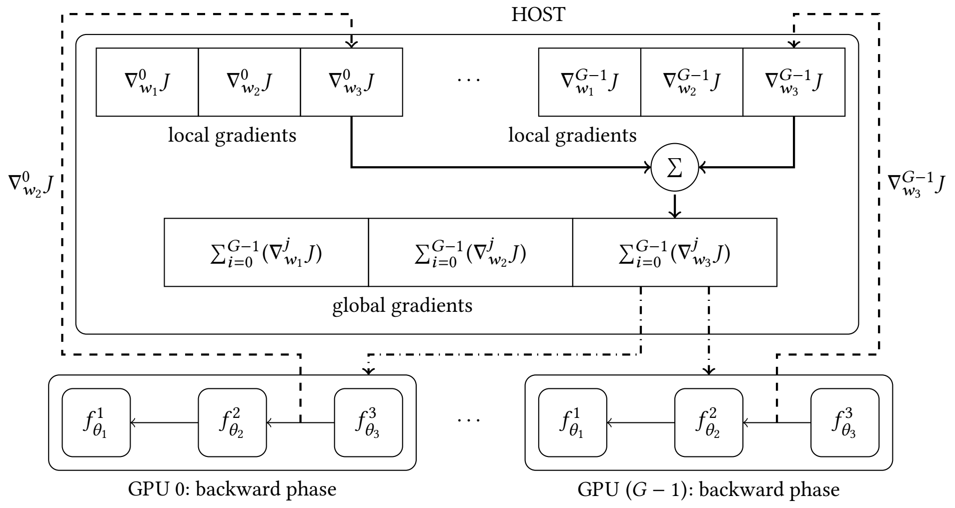


Figure 1: Communication pattern between host and GPUs during backward phase in CGDP training.

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

Step P-2: Each GPU performs the backward phase (explained below).

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

During the backward phase (Step P-2), gradients accumulations are performed on the 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 their learnable parameters in parallel. This is different from the typical data parallel training in which only the GPU 0 has the accumulated gradients and does the update phase. In other words, there is no parameter server in the CGDP training.

Step P-2 is an extension of the original back-propagation algorithm (Section 2.2). Five steps are added: Steps B-4-1, B-4-2, and B-4-3 after Step B-4; and Steps B-5-1 and B-5-2 after Step B-5. Let Q be a queue to store the layers for which backward computation has been completed. If a layer is in Q, its gradient has been sending to the host or accumulated into the global gradient on the host. Step P-2 for each GPU is as follows (the GPU stream is shown in parentheses):

Step B-1 (Default stream): Compute the gradient for the output layer:

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

Step B-2 (Default stream): Compute the gradient of the activation for 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 (Default stream): Compute the gradients of 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 (Default stream): 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-4-1 (Host): Synchronize the default stream with respect to the host.

Step B-4-2 (D2H stream): Send local gradients $\nabla^j_{\theta_l}J$ to the host, and call the callback function to accumulate the local gradients into the global gradient on the host. Push l to Q.

Step B-4-3 (H2D stream): For each layer k in Q, if all local gradients $\nabla^j_{\theta_k}J,\ j = 0, \ldots, (G-1)$ have been accumulated into the global gradient, broadcast the global gradient to all GPUs and remove k from Q.

Step B-5: Continue steps B-2 to B-4-3 until l reaches 1.

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

Step B-5-2 (Default stream): Synchronize the H2D stream with respect to the host.

The global gradient of a layer is the gradient accumulated from all local gradients of that layer from all GPUs. The synchronization step (Step B-4-1) is particularly important because it ensures that the local gradient of a layer is sent to the host after the gradient computation has finished.

Figure 1 illustrates the communication pattern between the host and GPUs during the backward phase of one iteration. On the host, for each GPU, there is a concurrent vector of the local gradients produced by the layers. There is also a concurrent vector of global gradients accumulated from the local gradients. The GPUs communicate directly with the CPUs to send gradients to the CPUs. Once a layer has computed its gradients with respect to its learnable parameters, the gradients are sent to the host, where they are accumulated into the global gradients (Step B-4-2). Gradient accumulation on the host is done in parallel using the OpenMP API. Once this gradient accumulation has been completed (by checking the existence of the layer in the queue Q), the global gradients are broadcasted back to all GPUs (Step B-4-3). At the same time, the next layer computes the other gradients (Steps B-2, B-3, B-4). Note that, once all layers have finished their computations, the completion of gradient accumulation for the last layer (or maybe the few last layers) has not finished yet. Hence, we need another step at the end of the backward phase to check for the completion by all layers and then broadcast the remaining global gradient(s) to the GPUs (Step B-5-1). Step B-5-2 ensures that all accumulated gradients are available in the GPUs before performing the update phase.

3.2 Cost model

We designed a cost model for the CGDP training and analyzed 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 in each network finish its computation at the same time. This means that it is sufficient to consider only the training for one GPU. Furthermore, we consider only the backward phase because we do not change the forward and update phases.

Given an l-layer FFN where f^i is its i-th layer, let t_{bp}^i be the time on the GPU for the layer's back-propagation (Steps B-1, B-2, B-3, B-4), and t_a^i be the time on the host for gradient accumulation, where t_a^i includes the synchronization time t_{as}^i (Step B-4-1) and accumulation time t_{aa}^i (Step B-4-2). Let t_{bc}^i be the time for broadcasting the accumulated gradient from the host to the GPUs (Step B-4-3).

In CGDP training, t_{aa}^i and t_{bc}^i overlap the next t_{bp}^j(s), $l \geq j > i$. In addition, t_{aa}^i and t_{aa}^j, $i \neq j$, overlap because they are handled by different processes in parallel.

Definition 3.1. The back-propagation time on a GPU for an l-layer FFN, T_{BP}, is defined by:

$$T_{BP} = \sum_{i=l}^{1}(t_{bp}^i)$$

Definition 3.2. The *total processing time* for a layer i in an l-layer FFN is defined by:

$$T_{BP}^i = \sum_{l}^{i}(t_{bp}^i + t_{as}^i) + t_{aa}^i + t_{bc}^i$$

Intuitively, the total processing time for a layer is the time from *the beginning of the backward phase* to the point where the layer's accumulated gradient is available in the GPU. Note that the definition of T_{BP}^i does not guarantee that, for $i < j$, the layer j will finish before the layer i during the backward phase.

Definition 3.3. The running time of the backward phase using CGDP training, T, is computed as

$$T = \max_{i=1,\ldots,l}(T_{BP}^i).$$

Intuitively, the running time of the backward phase depends on the slowest layer (the one with the longest total processing time).

Definition 3.4. The running time of the backward phase in the typical data parallel training, T', is computed as follows (in this case, there is no synchronization as only one stream, the default stream, is used; i.e., $t_{as} = 0$):

$$T' = T_{BP} + \sum_{i=l}^{1}(t_{aa}^i + t_{bc}^i).$$

Note that t_{bc} in T is the time for broadcasting the gradients from the host to every GPU, while t_{bc} in T' is the time for broadcasting the parameters from the server GPU to the other GPUs. For each layer, the numbers of the 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.

Overhead time is defined as the additional time for accumulating and exchanging gradients among GPUs in the backward phase. For the typical data parallel training, overhead time is computed as

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

. For the CGDP training, overhead time is non-trivial to compute. However, simplification by assuming that t_{aa} and t_{bc} of layers $l, l-1, \ldots, 2$ perfectly overlap the next t_{bp}(s) makes $T = T_{BP}^1$. The overhead time is then computed as

$$T_O = T - T_{BP} = \sum_{i=l}^{1}(t_{as}^i) + t_{aa}^1 + t_{bc}^1$$

The CGDP training is faster than the typical data parallel training if $T_O < T_O'$. If the layer 1 has a small number of parameters, then t_{aa} in T_O is approximately equal to t_{aa} in T_O'. Hence, $T_O < T_O'$ holds if $\sum_{i=l}^{1}(t_{as}^i) < \sum_{i=l}^{2}(t_{aa}^i + t_{bc}^i)$. This condition is easy to meet for several practical neural networks such as AlexNet, GoogLeNet-v1, and VGGNet-16.

4 CHUNK-SIZE OPTIMIZATION

As mentioned in the Introduction, we extend CGDP training by using chunks of layers to deal with very deep and "flat" neural networks in which the number of parameters for a layer is small and roughly equals the number for the other layers.

In the naive CGDP training, the gradients are sent to the host layer-by-layer, which is triggered by a synchronization using the default stream with respect to the host. In other words, the D2H stream waits for the computation of a layer in the default stream to finish. 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 describe how the naive CGDP training is extended by using chunks to reduce the effect of synchronizations on the performance of the backward phase and then present a runtime algorithm for automatically finding a good setting for the CGDP training with chunks.

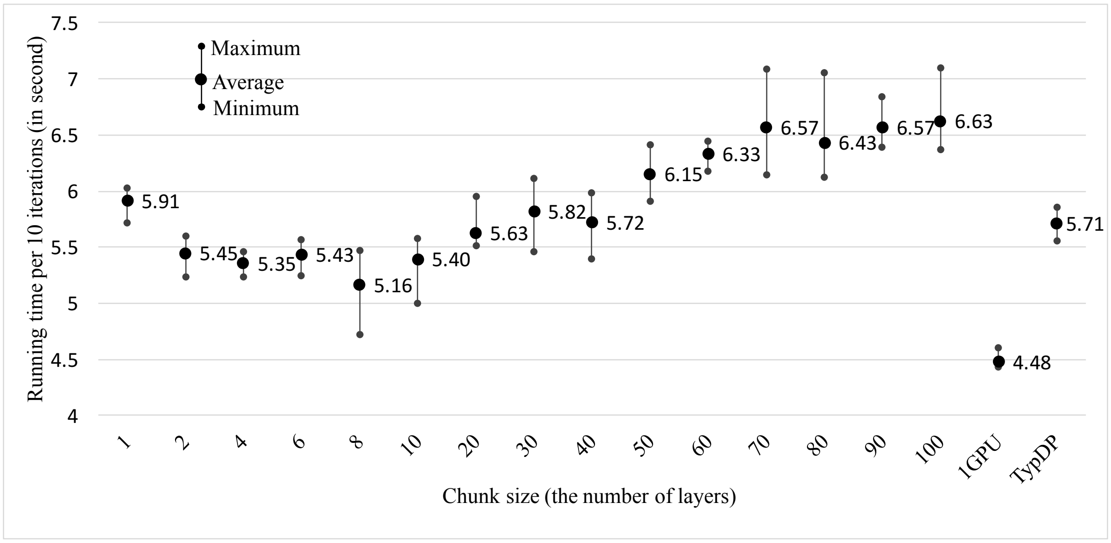


Figure 2: Running time for ten iterations of training ResNet-152 (minibatch size 16, 4 GPUs, no validation) for various chunk sizes. "TypDP" means using the typical data parallel training. The maximum, minimum, and average values were calculated for every ten iterations.

4.1 CGDP training using chunks

Because stream synchronization slows down the performance of the backward phase, it is better to perform synchronization after several layers have finished rather than after each layer has finished. In particular, it is best to optimize the $(\sum_{i=1}^{l}(t_{as}^{i}) + t_{aa}^{i} + t_{bc}^{i}))$ part of T.

We call a group of layers for which the gradients are sent to the host together *a chunk of layers*, and denote it by {}. Given a neural network, we can use multiple chunks with varying numbers of layers for CGDP training. For example, if a neural network has six 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 three 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 using three chunks is performed as follows: compute gradients for the layers of 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 the host, compute the 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 the host, compute the gradients for the layers of the chunk $\{f_{\theta_1}^{1}\}$, synchronize with the D2H stream to send the gradients of these layers to the host, and wait for all gradients to be available on the GPUs. In this example, only three synchronizations are needed using chunks while six synchronizations are needed without chunks.

There is a tradeoff between the number of synchronizations and the number of layers in a chunk. It is obvious that 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 for the 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 $(t_{aa}^{j}+t_{bc}^{j})$ for all layers j in the chunk to which layer i belongs. This makes it more difficult to optimize CGDP training.

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 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 the same space. □

4.2 Heuristic algorithm for finding chunks

Here we present a runtime algorithm for finding the chunk size that minimizes the running time for the backward phase. It is run for the first few training iterations to determine a good chunk size for reducing the running time for training. Two heuristic rules are used for determining how to expand the search space and how to stop the algorithm.

To limit the search space, here we consider only the case in which chunks have the same size except for the few 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 $(l \bmod k = 0)$, there are $(\frac{l}{k} - 1)$ chunks with size k, including layers from l to $(k+1)$, and there are k chunks with size 1, including the remaining layers from k to 1. If $(l \bmod k \neq 0)$, there are $\lfloor \frac{l}{k} \rfloor$ chunks with size k, including layers from l to $(l - k * \lfloor \frac{l}{k} \rfloor + 1)$, and there are $(l - k * \lfloor \frac{l}{k} \rfloor)$ chunks with size 1, including the remaining layers from $(l - k * \lfloor \frac{l}{k} \rfloor)$ to 1. The few last layers have a chunk size of 1 in order to reduce the effect of t_{aa}^{i} and t_{bc}^{i} on training overhead.

Figure 2 shows the running time for training ResNet-152 for chunk sizes from 1 to 100. It also shows the running times for single-GPU training and the typical data parallel training to illustrate the overhead of the CGDP training. The naive CGDP training (with chunks of size 1) was slower than the typical data parallel training. This is because ResNet-152 has many small layers (the number of parameters is small). Hence, the overhead of synchronizations is high, which results in the total overhead being high. Increasing the chunk size (chunk sizes 2, 4, 6, 8), gradually improved the results. However, using a large chunk size is not good due to the overhead of gradient accumulation. Compared to the running time for single-GPU training, the overhead time for multiple GPU training was very high, so there is potentially room for improvement.

Algorithm 1 shows our algorithm for finding a good chunk size for CGDP training with chunks. Assume that we train a neural network using N iterations. There are two user-defined parameters in the algorithm: `step` and `range`. The `step` parameter is used to heuristically determine how to expand the search space for chunk size and how to stop the algorithm. The `range` parameter is used to determine how to stop the algorithm and it is used together with the parameter `step`. Variable `chunk` is used to store the chunk size for the current iteration. It is the variable to be optimized. It is continually updated during the execution of the algorithm and is set to the value of variable `best_chunk` upon completion. Variable `best_chunk` holds the chunk size that results in the minimum running time, which is stored in a variable `lapse_min`. Variable `lapse` is the running time of the last `interval` iterations. At the beginning of training, variables `chunk`, `best_chunk`, and `lapse_min` are set to 1, 1, and $+\infty$, respectively (Line 1).

Our heuristic algorithm for finding a good chunk (Lines 7–25 in Algorithm 1) runs as follows. After each `interval` iteration, the algorithm is triggered. Although the value of `interval` can be changed, we use a fixed value of 10 here. The algorithm measures the running time for the `interval` iterations, and stores it in variable `lapse` (Line 13). If `lapse < lapse_min`, the values of `best_chunk` and `lapse_min` are updated to the current values (Lines 14–16). As mentioned, there are two heuristic rules in the algorithm. The first rule is used to expand the search space of the chunk size: `chunk := (chunk < step)?(chunk+1) : (chunk+step)` (Lines 18–22). The rule says that, at the beginning of the algorithm, if `chunk < step`, the value of the chunk size is gradually increased by 1. Otherwise, the chunk size is increased by `step`. This rule flexibly adjusts the search space of the algorithm. If `step` is large and close to the number of layers, most of the values for the chunk size are aggressively scanned. If it is small, big jumps in chunk size are made, and some values are ignored, which speeds up completion. The second rule is used to determine when to stop the algorithm; that is, `chunk ≥ best_chunk + step ∗ range` (Lines 8–10). Intuitively, once a `best_chunk` is found, the algorithm runs another `range` times. If there is no a better chunk size, the algorithm stops. Note that, if `best_chunk < step`, the algorithm runs another (`step − best_chunk + range`) times before determining whether to stop.

Algorithm 1 Runtime algorithm for finding a good chunk size

```
1:  procedure CGDP($N$, $step$, $range$)
2:      $chunk \leftarrow 1; best\_chunk \leftarrow 1; lapse\_min \leftarrow FLOAT\_MAX$
3:      $iter \leftarrow 1; interval \leftarrow 10; done \leftarrow FALSE$
4:      $start\_time \leftarrow$ System.currentTimeMillis()
5:      while $iter \leq N$ do                    ▷ Training for $N$ iterations
6:          CGDPTrainByChunk($chunk$)              ▷ See Section 4.1
7:          if ($iter$ mod $interval = 0$) and ($done \neq TRUE$) then
8:              if $chunk \geq best\_chunk + step * range$ then
9:                  $chunk \leftarrow best\_chunk$
10:                 $done \leftarrow TRUE$
11:             else
12:                 $end\_time \leftarrow$ Sys.currentTimeMillis()
13:                 $lapse \leftarrow (end\_time - start\_time)$
14:                 if $lapse < lapse\_min$ then
15:                     $lapse\_min \leftarrow lapse$
16:                     $best\_chunk \leftarrow chunk$
17:                 end if
18:                 if $chunk < step$ then
19:                     $chunk \leftarrow chunk + 1$
20:                 else
21:                     $chunk \leftarrow chunk + step$
22:                 end if
23:                 $start\_time \leftarrow$ Sys.currentTimeMillis()
24:             end if
25:         end if
26:         $iter \leftarrow iter + 1$
27:     end while
28:     return
29: end procedure
```

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

5 EXPERIMENTAL RESULTS

5.1 Configurations

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

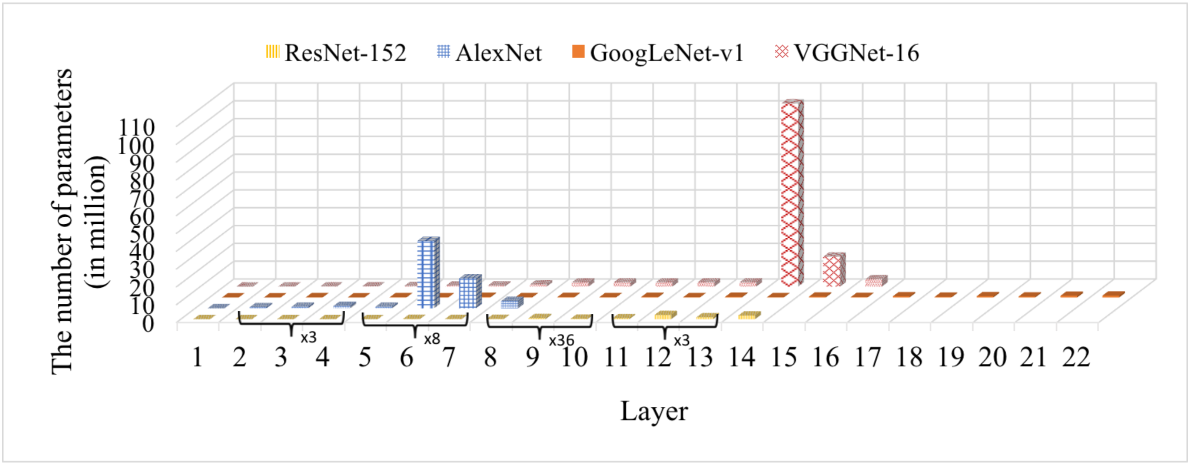


Figure 3: Number of parameters per layer for AlexNet, GoogLeNet-v1, VGGNet-16 and ResNet-152. Strings "x3", "x8", "x36", "x3" are used to indicate the number of blocks of layers for ResNet-152 only, e.g., there are three blocks of layers 2, 3, and 4.

We used four neural networks that are widely used in computer vision: AlexNet[1] [17], GoogLeNet-v1[2] (Inception-v1) [24], VGGNet-16[3] (model D) [22], 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. The distributions of parameters in the layers of these networks are shown in Figure 3. The layers in GoogLeNet-v1 and ResNet-152 have relatively the same size, and they are small, while the last few layers in AlexNet and VGGNet-16 are quite large compared to the other layers. The dataset used for training was a subset of the ImageNet ILSVRC2012 [20] database, which 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) used 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 running 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 for CGDP training

5.2.1 Running time for training. The scalability of CGDP training compared to the typical training was determined by analyzing the running time for training. Table 2 shows the running times for one training iteration with one, two, and four GPUs for AlexNet, GoogLeNet-v1, and VGGNet-16. The results for ResNet-152 are analyzed in detail in Section 5.3. The results in Table 2 indicated that TRL/Caffe is more scalable than BVLC/Caffe. When the number of GPUs was one, the running times for both frameworks were almost the same for each network. When the number of GPUs was four, TRL/Caffe was the fastest for all networks—in particular, it was 1.21, 1.04, and 1.21 times faster than BVLC/Caffe for AlexNet, GoogLeNet-v1, and VGGNet-16, respectively. This shows that TRL/Caffe consistently had a high efficiency ($\geq$ 90%). Our approach was the least effective for GoogLeNet-v1 and the most effective for VGGNet-16. This is because 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 number of parameters in GoogLeNet-v1 and VGGNet-16 are the least and the most, respectively, so we obtained corresponding results.

5.2.2 Communication overhead. Reducing communication overhead is our objective, and Table 3 shows the running time for every phase in training AlexNet on four 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 that in BVLC/Caffe. This is reasonable because TRL/Caffe needs to do work to invoke data copy functions and callback functions between two consecutive layers during the backward phase. As for GoogLeNet-v1 and VGGNet-16, the communication overheads of TRL/Caffe were the same as the one for AlexNet ($\approx$ 21.8μs). We thus do not show them here.

5.2.3 Time for accumulation on host. It is important to determine whether the accumulation on the host can be overlapped with

[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

Table 2: Running time for one training iteration (ms).

	AlexNet			GoogLeNet-v1			VGGNet-16		
no. of GPUs	1	2	4	1	2	4	1	2	4
BVLC/Caffe	157.2	174.3	202.7	140.0	151.7	163.4	345.6	383.7	445.6
TRL/Caffe	157.4	163.6	167.1	140.9	151.9	156.7	345.2	361.5	369.1

Table 3: Running time for phases in one iteration for AlexNet with four GPUs (in ms).

	broadcast	*forward*	*backward*	*broadcast remaining gradients from 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 4: Communication time between GPUs and CPUs, and accumulation time on CPUs for AlexNet with four GPUs.

	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

Table 5: Memory consumption in GPUs (in MB).

	AlexNet	GoogLeNet-v1	VGGNet-16
BVLC/Caffe	6863	6095	7274
TRL/Caffe	6359	5991	6191

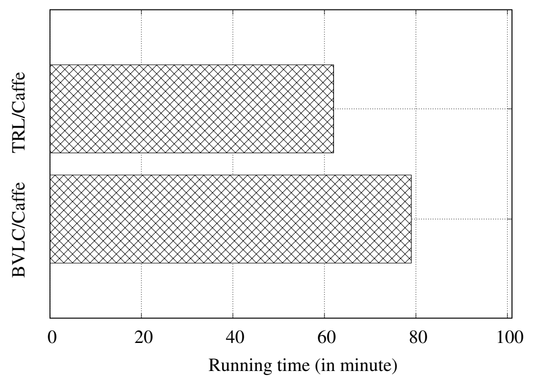
the computations on the GPUs. Table 4 shows the time for accumulation on the CPUs for each layers during the backward phase and the communication time between the GPUs and the CPUs for TRL/Caffe with AlexNet when using four GPUs. The communication and accumulation overlapped the backward phase. Note 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 ($\approx 7\mu$s).

5.2.4 Memory consumption. By offloading gradient accumulation onto the host, we can reduce memory consumption in the GPUs, which facilitates training with a larger batch size. Table 5 shows the maximum sizes of the memories allocated on the 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 that 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 the host. Accordingly, TRL/Caffe consumed more memory on the host. This is not a big problem because memory on host is generally cheaper and easier to increase than that on the 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.

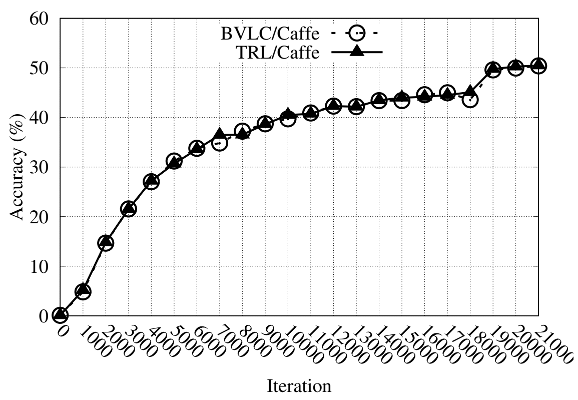
5.2.5 Long-term runs. We conducted experiments using long-term runs to evaluate the effectiveness of our approach for long-term runs. The objective was to train AlexNet to achieve 50% accuracy using four GPUs. TRL/Caffe took 62 minutes while BVLC/Caffe took 79 minutes (Fig. 4a). Both versions reached 50% accuracy at around iteration 20, 000 (21, 000 iterations in total) and had the same convergence curve (Fig. 4b). Note that this training included a testing phase, in which the 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.

5.3 Effectiveness of runtime algorithm

To test the runtime algorithm (Algorithm 1) we used ResNet-152 instead of Resnet-1001, which is too large to fit in our GPU memory, we used Resnet-152 instead. We trained ResNet-152 for 1000 iterations using four GPUs with a real world setting in which the validation phase was included. The maximum minibatch 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 stabilize the elapsed time stable. The effectiveness of each parameter in the runtime algorithm was examined.



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



(b) Accuracy per iteration in AlexNet training.

Figure 4: Long-term run for AlexNet training (21, 000 **iterations).**

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

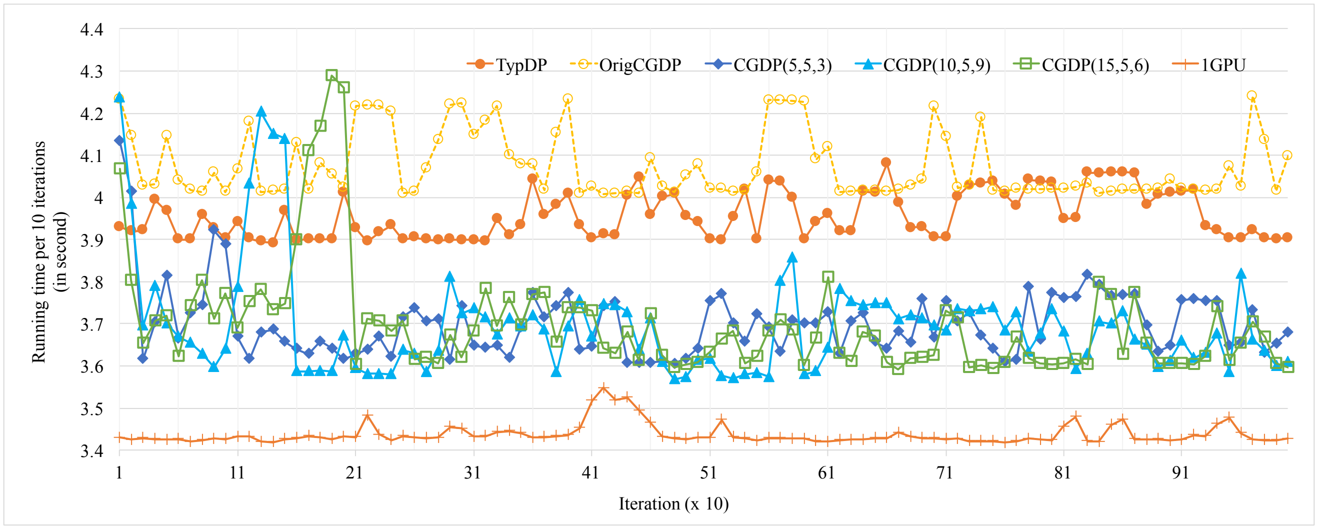
Figure 5b shows the results of extending the search space for a fixed value of step by increasing the value of range. A larger value for the chunk size was found: 20, when range was 10. Nevertheless, training with a chunk size of 20 had the same performance as the ones with other best chunk sizes (3, 6, or 9). In all cases, the runtime algorithm finished in at most 21 iterations, which shows that the runtime overhead of the algorithm was small because the whole training often had hundreds of thousands of iterations. Overall, CGDP training was about 1.07 times faster than the typical data parallel training for ResNet-152. For AlexNet, VGGNet-16, and GoogLeNet-v1, the runtime algorithm could not find a chunk (> 1) that produced a better result. This is because these neural networks have a small number of layers, so the effect of synchronization is small.

6 RELATED WORK

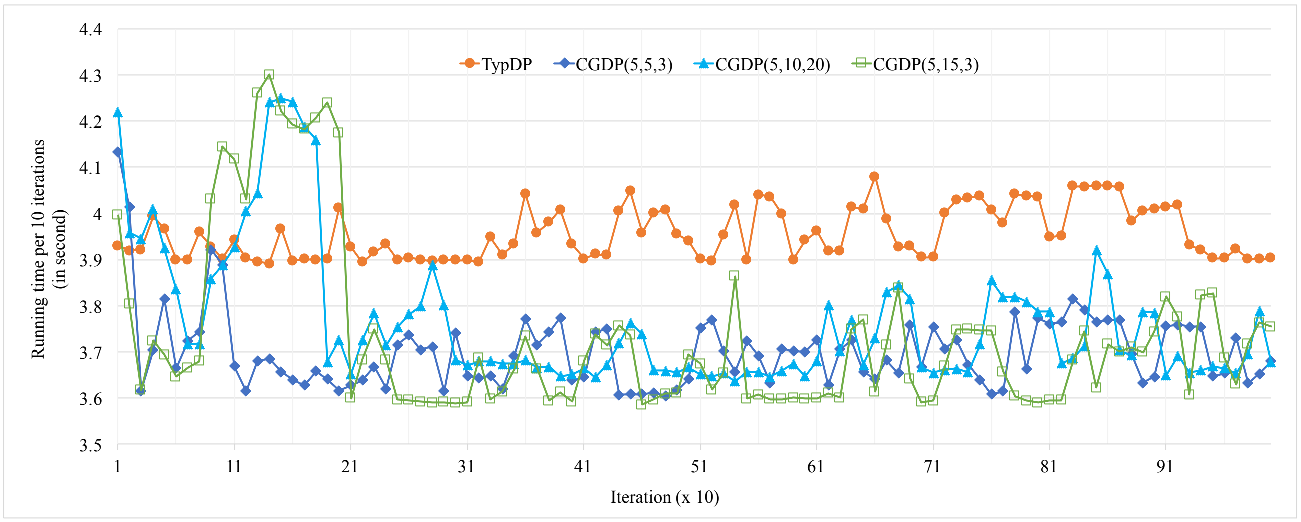
There are several ways to accelerate deep learning: data parallelism, model parallelism, and pipeline parallelism. Data parallelism is implemented in many frameworks such as Google's TensorFlow [5], Torch [8], and Microsoft's CNTK [25]. It is mainly used for deep convolutional neural networks. Model parallelism has been used for large-scale unsupervised learning [18]. Many distributed frameworks, such as MXNet [6], Mariana [28], the COTS HPC system [7], and the DistBelief software framework [9], support both data parallelism and model parallelism so that users can use either one. For single-machine training, both TensorFlow and MXNet support gradient accumulation on the host, but the accumulation is performed at the end of the back-propagation computation instead of layer-by-layer as in our CGDP training. 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 [23]. Pipeline optimization is used in Mariana [28]: a three-stage pipeline, 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 [26], 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 that is close to ours is one developed at Google [19] in which a method learns to predict a set of device placements for layers in a neural network. It is targeted at heterogeneous distributed environments with a mix of hardware devices such as CPUs, and GPUs. In our CGDP training, the layers are always computed in the GPUs, so device placement is not needed.

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



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



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

Figure 5: Effectiveness of parameters in runtime algorithm for ResNet-152 with minibatch size of 12 per GPU. Plotted are CGDP values for (`step`, `range`, `best_chunk`), where `best_chunk` is the best chunk size found by the algorithm. "TypDP" refers to the typical data parallel training. "OrigCGDP" refers to the naive CGDP training with the chunk size of 1.

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, 27], but it changes the learning accuracy, and it is sometimes difficult to exactly reproduce the result. Therefore, the asynchronous SGD is not our target.

7 CONCLUSION AND FUTURE WORK

Deep learning is an emerging tool for signal processing and is mainly sped up by accelerators such as GPUs. This means that powerful CPUs are redundant in deep learning. Some frameworks, such as TensorFlow and MXNet, have been utilizing CPUs for gradient accumulation, but the objective is to make the frameworks flexible instead of improving performance. We have shown here for the first time to our knowledge that utilizing free CPUs on a host accelerates GPU-based data parallel training of deep neural networks on a single machine. In our CGDP training approach, CPUs collect, accumulate, and broadcast gradients produced during the backward phase. The key idea is that those operations can be performed in parallel with the backward phase, resulting in a reduction in time for each training iteration. We also presented a cost model for data parallel training of neural networks and demonstrate its power by using it to identify a bottleneck in training very deep neural networks. Finally, we presented a runtime algorithm using simple heuristics to optimize one of the deepest neural networks, ResNet-152. Future work includes extending the CGDP training to support recurrent neural networks, which consist of cyclic links among layers. An additional mechanism is needed for such networks to determine which layer gradients are kept in the GPUs for the cyclic links, and when to send them to the host. Additionally, it is open to investigate what kind of computation is suitable for being offloaded onto the 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] Azalia Mirhoseini, Hieu Pham, Quoc Le, Mohammad Norouzi, Samy Bengio, Benoit Steiner, Yuefeng Zhou, Naveen Kumar, Rasmus Larsen, and Jeff Dean. 2017. Device Placement Optimization with Reinforcement Learning. https://arxiv.org/abs/1706.04972

[20] 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

[21] 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).

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

[23] 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.

[24] 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.

[25] 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.

[26] 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).

[27] 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.

[28] 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.