Vol.2017-HPC-159 No.8
2017/4/17

Accelerating Multi-GPU Deep Learning by Collecting and Accumulating Gradients on CPUs

Tung Le Duc[1,a] Taro Sekiyama[1,b] Yasushi Negishi[1,c] Haruki Imai[1,d]
Kiyokuni Kawachiya[1,e]

Abstract: Training a deep neural network includes thousands of iterations and has significant computational cost. Data parallelism over multiple GPUs has widely used in deep learning frameworks to accelerate the training phase. However, data parallel training has high overhead due to the exchange of a large number of learnable parameters among GPUs. In this paper, we propose a novel approach to optimizing the exchange of parameters in data parallel training by 1) collecting gradients during the backward phase of an iteration from the GPUs to the CPUs and 2) offloading the gradient accumulation from the GPUs to the CPUs. Three neural networks: AlexNet, GoogLeNet-v1, and the VGGNet 16 layer network (model D) were trained using ImageNet dataset on four Tesla P100 GPUs in a IBM POWER8 machine. The overhead of parameter exchange per iteration was reduced from tens of milliseconds to tens of microseconds for all networks. We achieved more than 90% weak scaling efficiency for all networks.

1. Introduction

Deep learning is an effective tool for solving complex problems such as ones in computer vision, speech recognition, and natural language processing. For example, deep learning has been successfully used to recognize objects in digital images. Deep learning blossomed in 2012 when a deep convolutional neural network (one with convolutional layers [1]) called AlexNet developed by Krizhevsky et al. [2] achieved outstanding image classification results in the ILSVRC-2012 competition with a top-5 test error rate of 15.3%.

Training a deep neural network means optimizing the network loss function by finding feasible values of learnable parameters called *weights* and *biases*. There are many such optimization algorithms—for example, the stochastic gradient descent (SGD) algorithm with momentum [3], Adam [4], and Adagrad [5]—and they usually update parameters iteratively. Such processes are called *iterations*. Each iteration generally includes three phases. In the *forward phase*, the value of the loss function is computed from the starting layer to the ending layer of a network. In the *backward phase*, the gradients with respect to the learnable parameters is computed in the reverse direction (i.e., from the ending layer to the starting layer). In the *updateParameter phase*, all the parameters are updated using the gradients.

1 IBM Research - Tokyo, 19-21, Nihonbashi Hakozaki-cho, Chuo-ku, Tokyo 103-8510, Japan
a) tung@jp.ibm.com
b) sekiym@jp.ibm.com
c) negishi@jp.ibm.com
d) imaihal@jp.ibm.com
e) kawatiya@jp.ibm.com

Data parallelism is preferable for training convolutional neural networks over big data due to its simplicity and the fact that GPU memory is sufficient to store a deep neural network. That is why most deep learning frameworks, such as Caffe [6], TensorFlow [7], Torch [8], and the Computational Network Toolkit (CNTK) [9], are based on data parallelism. Furthermore, it is easy to extend data parallelism on a single machine with multiple GPUs to a distributed environment (multiple machines) without heavy modifications. For example, two distributed deep learning frameworks are based on Caffe: CaffeOnSpark[*1] and DeepSpark [10].

However, naive data parallelism does not scale well due to communication overhead. The communication overhead is significant, especially when training neural networks having an enormous number of parameters with many GPUs. This is because the greater the number of parameters and the greater the number of GPUs, the greater the number of gradients to be exchanged among GPUs.

In this paper, we propose using a novel approach to scaling data parallelism. In our approach, gradients are collected and accumulated on CPUs layer-by-layer during the backward phase. The use of CPUs enables the power of advanced processors to be used to accelerate training. While our approach does not affect the learning accuracy or the performance of the forward and the backward computations on the GPUs, it hides most of the communication overhead in the data parallelism behind the backward phase of training. Our approach is particularly effective for convolutional neural networks. The layers in convolutional neural networks

*1 CaffeOnSpark: `https://github.com/yahoo/CaffeOnSpark` 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 take much time; furthermore, since collecting and accumulating the gradients of starting layers takes less time, they will be completed immediately after the backward phase with a very low overhead.

The main contribution of this work is the design, implementation, and analysis of an overlap mechanism that hides most of the communication overhead in data parallelism behind the backward phase of a training by utilizing CPUs. Our design is a coarse-grained optimization in the sense that it does not change the core of deep neural network training (forward, backward, and updateParameter phases). Hence, it can be applied to a wide range of deep learning frameworks. We implemented our idea in the Caffe [6] deep learning framework, which is widely used in computer vision and is compatible with new commits from the Caffe GitHub community. Testing using the ImageNet dataset [11] (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 between the CPUs and GPUs [12] showed that the communication overhead per learning iteration was reduced from tens of milliseconds to tens of microseconds for three state-of-the-art convolutional neural networks: AlexNet [2], GoogLeNet-v1 [13], and the VGGNet 16 layer network (model D) (VGGNet hereafter) [14]. In particular, we achieved more than 90% weak scaling efficiency for all three networks when four GPUs were used. For AlexNet in particular, our approach reduced the learning time from 79 minutes to 62 minutes with 50% accuracy. An interesting finding is that offloading the gradient accumulation to the CPUs increased the memory available on the GPUs, which faciliates training a network with many datasets—e.g., VGGNet-16Layers can be trained with using 10% more images.

This paper is organized as follows. Section 2 presents an overview of Caffe, a state-of-the-art deep learning framework in computer vision. Section 3 describes in detail our optimization for making Caffe more scalable by minimizing communication overhead. We focus on a single machine coupled with multiple GPUs. Experiment results for a real dataset (ImageNet [11]) are presented in Section 4. Related work is discussed in Section 5, and key points are summarized in Section 6.

2. Deep Learning Framework

2.1 Definition

A neural network is defined as a set of layers by using a plaintext modeling language. Layers are organized in a directed acyclic graph and each node in the graph represents a layer. Though layers can be organized as a cyclic graph, this paper only focuses on neural networks that are directed acyclic graph. Connections between layers are automatically inferred from their named inputs and outputs.

Algorithm 1 Sequential training algorithm using SGD

```
Require: Neural network $net$, solver $sv$.
Ensure: Neural parameters are updated.
1: $i \leftarrow 0$
2: while ($i < sv.iteration\_numbers()$) do
3:     loss ← net.$forward()$
4:     net.$backward()$
5:     sv.$updateParameter()$
6:     $i \leftarrow i + 1$
7: end while
```

A layer is the essence of a neural network. It takes one or more *b*inary *l*arge *ob*jects (*blobs*) as input and returns one or more blobs as output. Input blobs are called *bottom blobs* and output blobs are called *top blobs*. (Blobs will be discussed in more detail at the end of this section.) Inside a layer, there are three important routines: *setup*, *forward*, and *backward*. Setup is used to initialize the layer and its connections once the model is initialized. Forward is used to compute top blobs from bottom blobs. Backward is used to compute the gradient with respect to the bottom input given the gradient with respect to the top output. If a layer has learnable parameters, the gradient with respect to the parameters is computed and stored internally in order to update the parameters.

A blob is a multi-dimensional array used as a data structure to provide seamless synchronization capability between the CPUs and the GPUs. It serves as a unified memory interface for storing data as well as communicating between them. Blobs are used to hold data such as a batch of images, model parameters, and derivations for optimization.

Figure 1 shows a simplified version of the AlexNet neural network [2]. Only the layers with learnable parameters are shown. They comprise 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, in which network loss is computed. Blobs are represented by hexagons. Taking the layer “CONV2” as an example, it accepts bottom blob “conv1” as input and outputs top blob “conv2”.

2.2 Training neural networks

As explained in Introduction, training a deep neural network consists of many iterations of three phases: forward, backward, and updateParameter phases(see Algorithm 1). In general, the network loss value is computed in the forward phase, the gradients with respect to the learnable layer parameters are computed in the backward phase, and the parameters are updated in the updateParameter phase. Because the input dataset is usually too large to fit in the memory of a single machine or GPU device, the training is often done in multiple iterations, and a minibatch of data

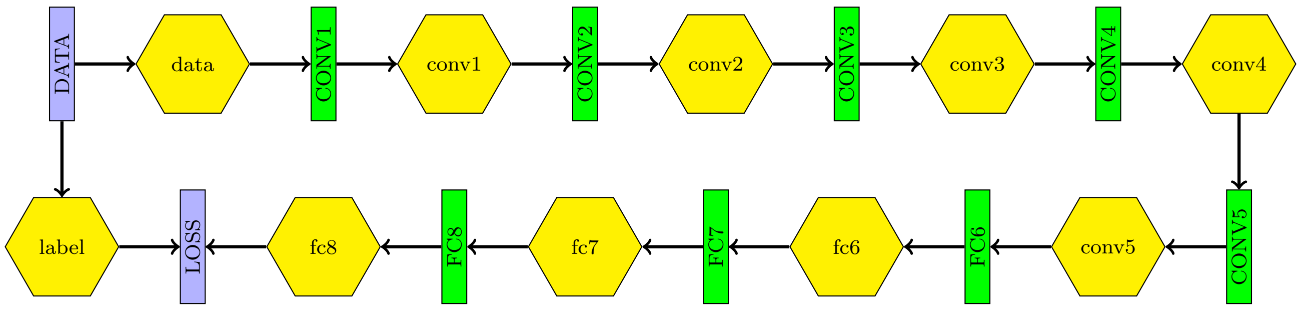


Fig. 1: AlexNet deep neural network

Algorithm 2 Forward algorithm in Caffe

```
Require: L network layers
Ensure: Network loss
 1: act[0] ← net.data()
 2: for $i = 1$ to $L$ do
 3:     act[$i$] ← net.layers[$i$].forward(act[$i - 1$])
 4: end for
 5: loss ← net.loss(atv[$L$], label)
 6: return loss
```

Algorithm 3 Backward algorithm in Caffe

```
Require: L activations of layers
Ensure: Layer gradients are computed
 1: grads[$L$] ← net.backward(act[$L$])
 2: for $i = L - 1$ to 1 do
 3:     grads[$i$] ← net.layers[$i$].backward(act[$i$], grads[$i + 1$])
 4: end for
```

records (e.g., images) is processed in each iteration.

Algorithm 2 corresponds to the forward phase of an iteration. Global blob *act* is used to store intermediate layer outputs (activations). First, a minibatch is read and transformed by the layer DATA (line 1) and stored in the first element of act. Then, forward computations are performed layer-by-layer (lines 2–4). Finally, the loss of the network is computed from the last activation (output) and the ground-truth label (line 5).

Algorithm 3 corresponds to the backward phase of an iteration. Global blob *grads* is used to store intermediate layer gradients. Backward computation is performed from the top to the bottom. First, the gradients are computed with respect to the output (line 1), and then they are computed with respect to the rest of the network layer-by-layer by using the chain-rule for gradients in mathematics.(lines 2–4).

The learnable parameters for the whole network (i.e, the parameters for all layers) are updated using the act and grads. Hyperparameters such as learning rate and momentum are also used to update the parameters. Such hyperparameters together with the number of iterations are defined in a solver. Some widely used solver algorithms include the SGD algorithm with momentum [3], Adam [4], and Adagrad [5].

2.3 Data parallelism for training

Algorithm 4 Data parallel algorithm for training

```
Require: Neural network net; solver sv.
Ensure: Neural network with updated parameters.
 1: $i \leftarrow 0$
 2: while ($i <$ sv.iteration_numbers()) do
 3:     broadcastParameters()
 4:     loss ← net.forward()
 5:     net.backward()
 6:     collectGradients()
 7:     if sv.is_root() then
 8:         sv.updateParameter()
 9:     end if
10:     $i \leftarrow i + 1$
11: end while
```

Data parallelism for neural network training on GPUs is quite straightforward. Algorithm 4 shows the data parallel training. At every iteration of training, each GPU trains the network by using a different minibatch. Then, one GPU (the root GPU) updates the parameters for the whole network (lines 7–9). More specifically, at the beginning of an iteration, the root GPU broadcasts its parameters to the other GPUs to ensure that every GPU has the same network parameters (line 3). The GPUs then perform their forward and backward phases using different minibatches to compute the gradients. The gradients are then collected and accumulated (line 6) in the root GPU where a parameter update is done. The number of gradients is equal to the number of learnable parameters because a gradient is computed for each parameter. In Caffe, data transfers and accumulations between GPUs are done in accordance with a GPU tree representing the optimized physical links between GPUs, which minimizes communication overhead.

Let us consider weak scaling of the data parallelism, in which the number of GPUs is increased while the size of the minibatch for each GPU remains unchanged. It is clear that the times for the forward and backward phases remains basically unchanged. However, the times for broadcasting the parameters and collecting the gradients increase with the number of GPUs due to communication overhead between GPUs. This results in poor scaling of the data parallelism.

3. Coarse-grained optimization of communication

In this section, we explain our main idea—optimizing data

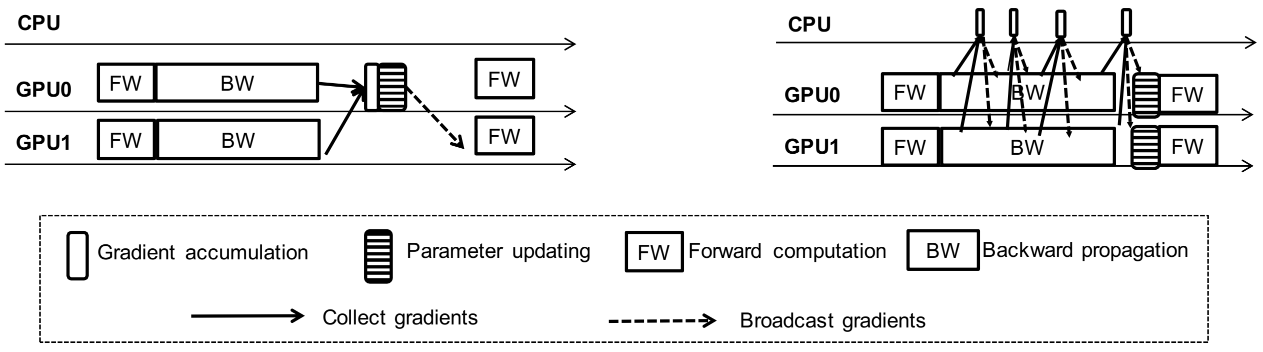


Fig. 2: Comparison of Caffe and proposed data parallel model

Algorithm 5 Our data parallel algorithm for training

```
Require: Neural network $net$; solver $sv$.
Ensure: Neural network with updated parameters.
1: $i \leftarrow 0$
2: $broadcastParameters()$
3: while ($i <$ sv.$iteration\_numbers()$) do
4:     loss ← net.$forward()$
5:     net.$backward\_()$
6:     $collectRemainingGradsFromCPU()$
7:     sv.$updateParameter()$
8:     $i \leftarrow i + 1$
9: end while
```

parallel training on multiple GPUs. Our optimization is coarse-grained in the sense that it is high level. We do not change the training of the layer computations or change solver algorithms. This means that every functionality related to deep learning (say, learning accuracy) is preserved. Furthermore, such a coarse-grained optimization can be easily applied to different deep learning frameworks in general.

Algorithm 5 corresponds to our data parallel algorithm for training. First, the GPUs synchronize their learnable parameters (line 2) to ensure that every GPU learns with the same network. For each training iteration, we only modify the backward phase (line 5). During the backward phase, the layer output blobs (gradients) are collected and accumulated on the CPUs, overlapping with the backward phase on the GPUs (Section 3.1). The gradients accumulated on CPUs are sent back to *every GPU*, also overlapping with the backward phase (Section 3.2). At the end of the backward phase, the remaining updated gradients are collected from the CPUs to the GPUs to ensure that all the gradients are sent to the GPUs (line 6). This may introduce some overhead, but in practice the overhead is very low because the ending layers in the backward phase are often convolutional layers that have a small number of learnable parameters (details are discussed in Section 4). After receiving the gradients, the GPUs update their own parameters (line 7). Because the GPUs train using the same learnable parameters and update the parameters with the same gradients (received from the CPUs), every GPU holds the same learnable parameters after the update, and it can start a new iteration without communicating with another GPUs.

Figure 2 illustrates the difference between the conventinal

Algorithm 6 Our backward propagation algorithm

```
Require: $L$ activations of layers; $u$ updated_layer
Ensure: Layer gradients are computed
1: grads[$L$] ← net.$backward$(act[$L$])
2: $u \leftarrow L - 1$
3: for $i = L - 1$ to 1 do
4:     grads[$i$] ← net.layers[$i$].$backward$(act[$i$], grads[$i + 1$])
5:     MemcpyD2HAsync(p_grads[$i$], grads[$i$], d2h_stream)
6:     accGradsCallback($i$, d2h_stream)
7:     if ($u \geq 0$) & (isUpdated($u$)) then
8:         MemcpyH2DAsync(grads[$u$], g_grads[$u$], h2d_stream)
9:         $u \leftarrow u - 1$
10:    end if
11: end for
```

data parallel algorithm (left hand side) and our proposed algorithm (right hand side). While the conventional data parallel algorithm doesnot use CPUs for gradient accumulation, our algorithm use the CPUs to accumulate the gradients layer-by-layer during the backward propagation (details are discussed in the next section). In our algorithm, every GPU does the same computation including forward computation, backward computation, and parameter updating. Hence, after parameter updating, there is no need to broadcast the updated parameter to every GPUs. This is only necessary to be done once at the beginning of the program.

3.1 Collecting gradients during backward propagation

In this section, we describe our backward algorithm for collecting and accumulating layer gradients during backward propagation. This algorithm is based on the observation that the layer gradients remain unchanged during backward propagation, so there is no need to postpone gradient accumulation until the end. Furthermore, to avoid interrupting the backward propagation on the GPUs, the gradients are sent to the CPUs, where they are processed as well. The use of different streams in the GPU programming means that the computations on the CPUs and GPUs and the communications between them overlap completely, thereby minimizing the communication overhead in data parallelism.

Algorithm 6 corresponds to our backward propagation algorithm. To accumulate the gradients on the CPUs, two parallel vectors of blobs need to be maintained on the CPUs:

Algorithm 7 Gradient accumulation on CPUs

```
Require: Layer index $i$
1: idx ← criticals[$i$]→pop()
2: #pragma omp parallel for
3: for $j$ = 0 to p_grads[$i$].size() - 1 do
4:     g_grads[$i$][$j$] ← g_grads[$i$][$j$] + p_grads[$i$][$j$]
5: end for
6: idx ← idx + 1
7: criticals[$i$]→push(idx)
```

one for partial gradients and one for global gradients. On each GPU, there is a blob for partial layer gradients (grads, and there is a copy of it on the CPUs (p_grads). The global gradients are used to store the accumulation results over partial gradients from different GPUs. Hence, the blobs for partial gradients and global gradients have the same size.

The backward propagation runs as follows. After backward propagation of layer i on a GPU has been completed, the partial gradient of the layer is copied from the GPU to the CPUs (line 5), and a callback function on the CPUs is called to accumulate the gradients (line 6). Then, whether layer u has been processed on the CPUs is checked. If it has been processed, each GPU copies the global gradient of u on the CPUs to the partial gradient of u on that GPU and updates u (lines 7–10). The copy from the GPUs to the CPUs and the callback are called with the same stream to ensure the correct order between them. Because the copies from the CPUs to the GPUs and from the GPUs to the CPUs are asynchronous and belong to two different streams, they completely overlap, and the next layer's backward computation will start immediately upon completion of the copy calls.

3.2 Accumulating gradients on CPUs

Algorithm 7 corresponds to the algorithm used to accumulate gradients on the CPUs. For each GPU, there is a CPU thread to handle the gradients for that GPU. Because threads update to the same location in the global gradient blob for the a particular layer, they need to be synchronized. This is done by maintaining a global array of blocking queues (criticals in Algorithm 7). There is one queue for each layer, each queue always contains one integer value. At the beginning of each iteration, these integer values are initialized to 0. One thread enters the critical section by obtaining the element in the queue of the target layer (line 1). It then accumulates the partial gradient for the global gradient by using an OpenMP parallel loop. Finally, it updates the integer value and pushes it back to the queue. When *all* threads have finished accumulating their partial gradients, the integer value in each queue is equal to the number of GPUs. This integer value is also used in the function isUpdate(u) in Algorithm 6 to check whether the gradients for layer u have been processed on the CPUs.

4. Experimental results

In this section, we describe the experimental results for the original version of Caffe [6]—hereafter, *BVLC/Caffe* to distinguish it—and for the version with our idea applied *TRL/Caffe*. A comparision shows that our approach reduces training time during training while preserving training accuracy.

Machine specifications

CPU	two 4GHz 10-core POWER8 processors 8 SMTs per core
CPU memory	512 GB
GPU	4 NVIDIA Tesla P100 GPUs
GPU memory	16 GB
Bandwidth	NVLink between GPUs and CPUs 80 GB/s duplex btwn GPUs 0 and 1 80 GB/s duplex btwn GPUs 2 and 3 80 GB/s duplex btwn CPUs and GPUs

Software versions

CUDA Toolkit	8.0.44
cuDNN	5.1.5 (the latest as of 30 Aug. 2016)
gcc	5.4.0
BVLC/Caffe	commit `b2982c7` (the latest as of 30 Aug. 2016)

Table 1: Machine specifications and software versions

	Layers	Parameters (million)	Minibatch size / GPU
AlexNet	8	60	256
GoogLeNet	22	13	64
VGGNet	16	138	32

Table 2: Neural networks and experimental settings

4.1 Experimental environment

The experiments were run on an IBM POWER8 machine [12] (see Table 1). The CUDA Toolkit is a development environment for building GPU-accelerated programs, and cuDNN is a state-of-the-art library for primitives used in deep neural networks.

To determine the effect of our optimization, we measured the training time and memory consumption during training, using three neural networks widely used in computer vision: AlexNet*2 [2], GoogLeNet*3 (Inception-v1) [13], and VGGNet*4 (model D) [14]. Table 2 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. We investigated whether our optimization changes the training results by examining the network outputs after training. The training dataset was a subset of ImageNet, which contains the 1000 categories and 1.2 million images and was used for the ILSVRC2012 challenge [11].

There is a performance problem with BVLC/Caffe that should be kept in mind. Briefly, although BVLC/Caffe strictly follows a data parallel model for training on multiple GPUs, in practice some of the GPUs are blocked during training, especially in the convolutional layers, making the training time longer than expected. This problem is due to

*2 AlexNet's network definition: `https://github.com/BVLC/caffe/tree/master/models/bvlc_alexnet`
*3 GoogLeNet's network definition: `https://github.com/BVLC/caffe/tree/master/models/bvlc_googlenet`
*4 VGGNet-16Layers-D's network definition: `https://gist.github.com/ksimonyan/211839e770f7b538e2d8`

the use of an empty kernel to synchronize between the non-default threads in the GPUs. To determine whether our optimization is effective even when the GPUs are utilized fully, we applied a pull request*5 that corrected the problem and used the corrected version—*BVLC/Caffe2*—for comparison; the implementation of TRL/Caffe is based on BVLC/Caffe2. In what follows, to show that BVLC/Caffe2 is inherently faster than BVLC/Caffe, we describe the results of all three versions BVLC/Caffe, BVLC/Caffe2, and 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.

4.2 Results

4.2.1 Execution time for training

Figure 3 shows the execution times for one training iteration with one, two, and four GPUs for AlexNet, GoogLeNet, and VGGNet. The results indicate that TRL/Caffe is more scalable than BVLC/Caffe and BVLC/Caffe2. When the number 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/Caffe2 for AlexNet, GoogLeNet, and VGGNet, respectively. These results show that our approach was the least effective for GoogLeNet and the most effective for VGGNet, 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 and VGGNet are the least and the most, respectively, so we obtained corresponding results.

We also conducted experiments using long-term runs to investigate whether our approach is also effective for long-term runs. The experiments are to to train AlexNet to achieve 50% accuracy The three versions reached 50% accuracy at around iteration $20,000$ ($21,000$ iterations in total). TRL/Caffe took 62 minutes while BVLC/Caffe and BVLC/Caffe2 take 94 and 79 minutes, respectively. This result shows that TRL/Caffe was also the fastest in long-term runs. Note that this training included a testing phase, in which a testing iteration was simply a forward computation using validation data 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.

4.2.2 Weak scaling

Figure 4 shows the weak scaling efficiency for three neural networks. For deep neural network training, weak scaling is more important than strong scaling when the objective is to train with as much data as possible. It is easy to see that TRL/Caffe consistently had a high efficiency ($\geq 90\%$). The efficiencies of BVLC/Caffe and BVLC/Caffe2 dropped substantially when the number of GPUs was increased from 2 to 4.

4.2.3 Communication overhead

Reducing communication overhead is our objective, and Table 3 shows the execution time by phase or AlexNet. We ignore the BVLC/Caffe version because it has too much overhead that is not caused by data parallelism itself. We can see that, for BVLC/Caffe2, 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 are hidden behind the backward phase, and the communication overhead is 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 in BVLC/Caffe2. This is reasonable because TRL/Caffe needs to do some works to invoke data copy functions and callback functions between layers during backward propagation. The communication overheads of TRL/Caffe for GoogLeNet and VGGNet were the same as the one in AlexNet ($\approx 21.8\mu$s). Hence we do not show them here.

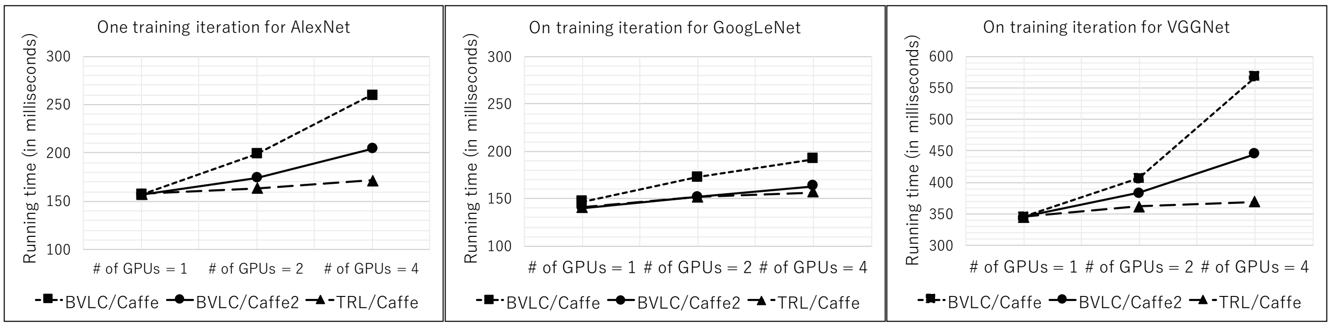
4.2.4 Time for accumulation on CPU

Table 4 shows the time for accumulation on the CPUs for each layers during backward propagation and the communication time between the GPUs and the CPU for TRL/Caffe with AlexNet (its architecture is shown in Figure 1). The communication and accumulation overlapped backward propagation. Note that in backward propagation, 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 backward propagation time. Furthermore, because the ending layer in backward propagation, layer 1, has a small number of parameters, the overhead for sending the gradients of this layer to the GPUs is very small ($\approx 7\mu$s).

4.2.5 Preservation of accuracy

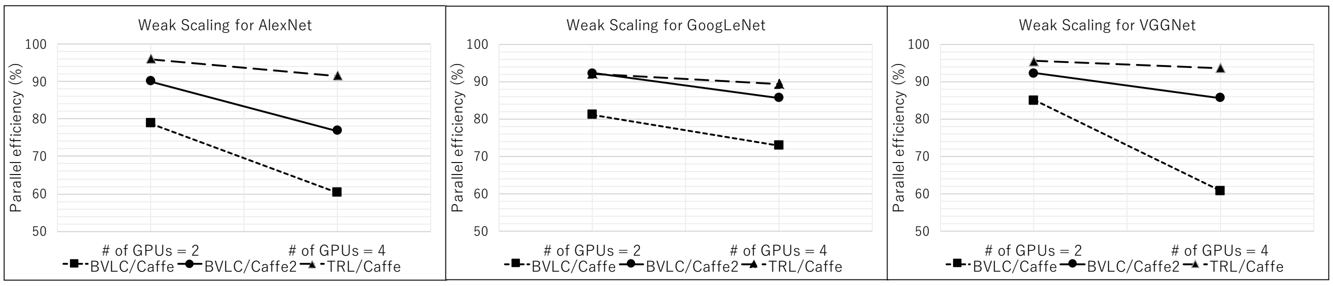
As shown above, our approach preserves training accuracy while reducing computation time and GPU memory usage. We confirmed this by examining the results of forward computation for VGGNet during training; if the results of BVLC/Caffe and TRL/Caffe are identical, their training algorithms set the learnable parameters to the same values. The reason why we use VGGNet is that it does not depend on layers, such as dropout [16], with probabilistic behavior and so the results of forward computation do not change as long as the parameters have the same values. We ran 500 training iterations with BVLC/Caffe, BVLC/Caffe2, and TRL/Caffe, checked the outputs of forward computation once every 20 iterations, and observed that they were the same among all Caffe versions. Even for AlexNet, for which the initial learnable parameters were randomly generated, we observed almost the same accuracy per iteration between

*5 PR#4386: `https://github.com/BVLC/caffe/pull/4368`



(a) AlexNet (b) GoogLeNet (c) VGGNet

Fig. 3: Execution time for one training iteration for three neural networks



(a) AlexNet (b) GoogLeNet (c) VGGNet

Fig. 4: Weak scaling for one training iteration with respect to the number of GPUs for three neural networks.

BVLC/Caffe2 and TRL/Caffe.

5. Related work

There are several ways to parallelize deep neural networks: data parallelism, model parallelism, and pipeline parallelism. Data parallelism is implemented in many frameworks such as Google's TensorFlow [7], Torch [8], and Microsoft's CNTK [9]. It is mainly used for deep convolutional neural networks. Model parallelism has been used for large-scale unsupervised learning [17]. Many distributed frameworks, such as MXNet [18], Mariana [19], COTS HPC system [20], and DistBelief [21], etc., support both data parallelism and model parallelism so that users can choose either one to use. Krizhevsky proposed a hybrid parallelism [22], 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 [19]: 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 [24], a distributed deep learning framework, in which there is overlap between backward computation and communication between 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.

6. Conclusion

Accelerating deep learning frameworks is important in designing neural networks. In this paper, we have presented a novel approach that combines the advantages of GPUs, CPUs, and fast links between them to speed up data parallel training of deep neural networks. A key feature of our approach is that it is coarse-grained in the sense that it is independent of the forward and backward computation of layers during training. Hence, it works well with the state-of-the-art training techniques that are being actively developed by the Caffe community. Furthermore, our approach is applicalbe to other data-parallelism-based frameworks. Experimental results for state-of-the-art deep neural networks and the latest GPUs showed that the communication overhead with our approach is very low (microsecond scale). Our approach consistently achieved more than 90% weak scaling efficiency for the three networks used, and it significantly reduced the training time for long-term runs with large datasets (e.g., ImageNet).

References

[1] K. Fukushima, "Neocognitron: A Hierarchical Neural Network Capable of Visual Pattern Recognition," *Neural Networks*, vol. 1, no. 2, pp. 119–130, 1988.
[2] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet

	broadcast	forward	backward	collectRemainingGradsFromCPU	grad-acc	update-param	total time
BVLC/Caffe2	20	50.6	104	N/A	23	5.1	202.7
TRL/Caffe	N/A	51	111	21.8μs	N/A	5.1	167.1

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

	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 4: Communication time between GPUs and CPUs, and accumulation time on CPUs

Classification with Deep Convolutional Neural Networks," in *International Conference on Neural Information Processing Systems*, 2012, pp. 1097–1105.

[3] N. Qian, "On the Momentum Term in Gradient Descent Learning Algorithms," *Neural networks*, vol. 12, no. 1, pp. 145–151, 1999.

[4] D. Kingma and J. Ba, "Adam: A Method for Stochastic Optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[5] J. Duchi, E. Hazan, and Y. Singer, "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization," *Journal of Machine Learning Research*, vol. 12, pp. 2121–2159, Jul. 2011.

[6] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional Architecture for Fast Feature Embedding," *arXiv preprint arXiv:1408.5093*, 2014.

[7] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: http://tensorflow.org/

[8] R. Collobert, K. Kavukcuoglu, and C. Farabet, "Torch7: A matlab-like environment for machine learning," in *BigLearn, NIPS Workshop*, 2011.

[9] D. Yu, A. Eversole, M. Seltzer, K. Yao, O. Kuchaiev, Y. Zhang, F. Seide, Z. Huang, B. Guenter, H. Wang, J. Droppo, G. Zweig, C. Rossbach, J. Gao, A. Stolcke, J. Currey, M. Slaney, G. Chen, A. Agarwal, C. Basoglu, M. Padmilac, A. Kamenev, V. Ivanov, S. Cypher, H. Parthasarathi, B. Mitra, B. Peng, and X. Huang, "An Introduction to Computational Networks and the Computational Network Toolkit," Tech. Rep., October 2014.

[10] H. Kim, J. Park, J. Jang, and S. Yoon, "DeepSpark: Spark-Based Deep Learning Supporting Asynchronous Updates and Caffe Compatibility," *arXiv preprint arXiv:602.08191*, 2016.

[11] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, "ImageNet Large Scale Visual Recognition Challenge," *International Journal of Computer Vision*, vol. 115, no. 3, pp. 211–252, 2015.

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

[13] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. E. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *IEEE Conference on Computer Vision and Pattern Recognition*, 2015, pp. 1–9.

[14] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[15] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, "cuDNN: Efficient Primitives for Deep Learning," *arXiv preprint arXiv:1410.0759*, 2014.

[16] N. Srivastava, G. E. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: a simple way to prevent neural networks from overfitting," *Journal of Machine Learning Research*, vol. 15, no. 1, pp. 1929–1958, 2014.

[17] Q. Le, M. Ranzato, R. Monga, M. Devin, K. Chen, G. Corrado, J. Dean, and A. Ng, "Building high-level features using large scale unsupervised learning," in *International Conference in Machine Learning*, 2012.

[18] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang, "Mxnet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems," *arXiv preprint arXiv:1512.01274*, 2015.

[19] Y. Zou, X. Jin, Y. Li, Z. Guo, E. Wang, and B. Xiao, "Mariana: Tencent Deep Learning Platform and Its Applications," *Proceedings of VLDB Endow.*, vol. 7, no. 13, pp. 1772–1777, Aug. 2014.

[20] A. Coates, B. Huval, T. Wang, D. Wu, B. Catanzaro, and N. Andrew, "Deep Learning with COTS HPC Systems," in *International Conference on Machine Learning*. JMLR Workshop and Conference Proceedings, 2013, pp. 1337–1345.

[21] J. Dean, G. S. Corrado, R. Monga, K. Chen, M. Devin, Q. V. Le, M. Z. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang, and A. Y. Ng, "Large scale distributed deep networks," in *International Conference on Neural Information Processing Systems*, 2012, pp. 1232–1240.

[22] A. Krizhevsky, "One Weird Trick for Parallelizing Convolutional Neural Networks," *arXiv preprint arXiv:1404.5997v2*, 2014.

[23] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to Sequence Learning with Neural Networks," in *International Conference on Neural Information Processing Systems*, 2014, pp. 3104–3112.

[24] H. Zhang, Z. Hu, J. Wei, P. Xie, G. Kim, Q. Ho, and E. Xing, "Poseidon: A system architecture for efficient gpu-based deep learning on multiple machines," *arXiv preprint arXiv:1512.06216*, 2015.

[25] W. Zhang, S. Gupta, X. Lian, and J. Liu, "Staleness-aware Async-SGD for Distributed Deep Learning," *arXiv preprint arXiv:1511.05950*, 2015.

[26] Q. Ho, J. Cipar, H. Cui, S. Lee, J. K. Kim, P. B. Gibbons, G. A. Gibson, G. Ganger, and E. P. Xing, "More Effective Distributed ML via a Stale Synchronous Parallel Parameter Server," in *International Conference on Neural Information Processing Systems*, 2013, pp. 1223–1231.