GeePS: Scalable deep learning on distributed GPUs with a GPU-specialized parameter server

Henggang Cui, Hao Zhang, Gregory R. Ganger, Phillip B. Gibbons, Eric P. Xing

Carnegie Mellon University

Abstract

Large-scale deep learning requires huge computational resources to train a multi-layer neural network. Recent systems propose using 100s to 1000s of machines to train networks with tens of layers and billions of connections. While the computation involved can be done more efficiently on GPUs than on more traditional CPU cores, training such networks on a single GPU is too slow and training on distributed GPUs can be inefficient, due to data movement overheads, GPU stalls, and limited GPU memory. This paper describes a new parameter server, called GeePS, that supports scalable deep learning across GPUs distributed among multiple machines, overcoming these obstacles. We show that GeePS enables a state-of-the-art single-node GPU implementation to scale well, such as to 13 times the number of training images processed per second on 16 machines (relative to the original optimized single-node code). Moreover, GeePS achieves a higher training throughput with just four GPU machines than that a state-of-the-art CPU-only system achieves with 108 machines.

1. Introduction

Large-scale deep learning is emerging as a primary machine learning approach for important, challenging problems such as image classification [7, 14, 23, 33] and speech recognition [13, 18]. In deep learning, large multi-layer neural networks are trained without pre-conceived models to learn complex features from raw input data, such as the pixels of labeled images. Given sufficient training data and computing power, deep learning approaches far outperform other approaches for such tasks.

The computation required, however, is substantial—prior studies have reported that satisfactory accuracy requires training large (billion-plus connection) neural networks on

100s or 1000s of servers for days [7, 14]. Neural network training is known to map well to GPUs [8, 23], but it has been argued that this approach is only efficient for smaller scale neural networks that can fit on GPUs attached to a single machine [7]. The challenges of limited GPU memory and inter-machine communication have been identified as major limitations.

This paper describes GeePS, a parameter server system specialized for scaling deep learning applications across GPUs distributed among multiple server machines. Like previous CPU-based parameter servers [10, 24], GeePS handles the synchronization and communication complexities associated with sharing the model parameters being learned (the weights on the connections, for neural networks) across parallel workers. Unlike such previous systems, GeePS performs a number of optimizations specially tailored to making efficient use of GPUs, including pre-built indexes for "gathering" the parameter values being updated in order to enable parallel updates of many model parameters in the GPU, along with GPU-friendly caching, data staging, and memory management techniques.

GeePS supports data-parallel model training, in which the input data is partitioned among workers on different machines that collectively update shared model parameters (that themselves are sharded across machines). This avoids the excessive communication delays that would arise in model-parallel approaches, in which the model parameters are partitioned among the workers on different machines, given the rich dependency structure of neural networks [33]. Data-parallel approaches are limited by the desire to fit the entire model in each worker's memory, and, as observed by prior work [7, 9, 14, 22, 30, 33], this would seem to imply that GPU-based systems (with their limited GPU memory) are suited only for relatively small neural networks. GeePS overcomes this apparent limitation by assuming control over memory management and placement, and carefully orchestrating data movement between CPU and GPU memory based on its observation of the access patterns at each layer of the neural network.

Experiments show that single-GPU codes can be easily modified to run with GeePS and obtain good scalable performance across multiple GPUs. For example, by modifying

Caffe [21], a state-of-the-art open-source system for deep learning on a single GPU, to store its data in GeePS, we can improve Caffe's training throughput (images per second) by 13× using 16 machines. Using GeePS, less than 8% of the GPU's time is lost to stalls (e.g., for communication, synchronization, and data movement), as compared to 65% when using an efficient CPU-based parameter server implementation. In terms of image classification accuracy, GeePS's rate of improvement on 16 machines is 8× faster than the single-GPU optimized Caffe's. The training throughput achieved with just four GPU machines beats that reported recently for a state-of-the-art 108-machine CPU-only system (ProjectAdam) [7], and the accuracy improvement with just 16 GPU machines is 4× faster than what was reported for a 58-machine ProjectAdam configuration. Experiments with video classification via recurrent neural networks show similar results. Interestingly, in contrast with recent work [7, 19, 24], we find that for deep learning on GPUs, BSP-style execution leads to faster accuracy improvements than more asynchronous parameter consistency models, because the negative impact of staleness outweighs the benefits of reduced communication delays.

Experiments also confirm the efficacy of GeePS's support for data-parallel training of very large neural networks on GPUs. For example, results are shown for a 20 GB neural network (5.6 billion connections) trained on GPUs with only 5 GB memory, with the larger CPU memory holding most of the parameters and intermediate layer state most of the time. By moving data between CPU memory and GPU memory in the background, GeePS is able to keep the GPU engines busy without suffering a significant decrease in training throughput relative to the case of all data fitting into GPU memory.

This paper makes three primary contributions. First, it describes the first GPU-specialized parameter server design and the changes needed to achieve efficient data-parallel multi-machine deep learning with GPUs. Second, it reports on large-scale experiments showing that GeePS indeed supports scalable data parallel execution via a parameter server, in contrast to previous expectations [7]. Third, it introduces new parameter server support for enabling such data-parallel deep learning on GPUs even when models are too big to fit in GPU memory, by explicitly managing GPU memory as a cache for parameters and intermediate layer state.

The remainder of this paper is organized as follows. Section 2 motivates GeePS's design with background on deep learning, GPU architecture, and parameter servers for ML. Section 3 describes how GeePS's design differs from previous CPU-based parameter server systems. Section 4 describes the GeePS implementation. Section 5 presents results from deep learning experiments with models of various sizes, including comparison to a state-of-the-art CPU-based parameter server. Section 6 discusses additional related work.

2. High performance deep learning

This section briefly describes deep learning, using image classification as a concrete example, and common approaches to achieving good performance by using GPUs or by parallelizing over 100s of traditional CPU-based machines with a parameter server architecture. Our goal is to enable the two approaches to be combined, with a parameter server system that effectively supports parallelizing over multiple distributed GPUs.

2.1 Deep learning

In deep learning, the ML programmer/user does not specify which specific features of the raw input data correlate with the outcomes being associated. Instead, the ML algorithm determines which features correlate most strongly by training a neural network with a large number of hidden layers [5], which consists of a layered network of nodes and edges (connections), as depicted in Figure 1.

Because deep learning is somewhat difficult to describe in the abstract, this section instead does so by describing how it works for the specific case of image classification [7, 14, 23, 28] and video classification [16, 34]. An image classification network classifies images (raw pixel maps) into pre-defined labels and is trained using a set of training images with known labels. The video classification network classifies videos (a sequence of image frames) into pre-defined labels, and often uses an image classification network as a submodule. This subsection describes an image classification network first and then describes how a video classification network can be built on it.
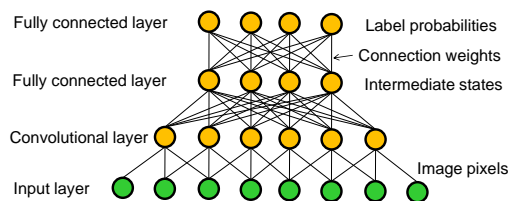

Figure 1. A convolutional neural network, with one convolutional layer and two fully connected layers.

The image classification task often uses a type of model called a *convolutional neural network* (CNN). The first layer of the nodes (input of the network) are the raw pixels of the input image, and the last layer of the nodes (output of the network) are the probabilities that this image should be assigned to each label. The nodes in the middle are intermediate states. To classify an image using such a neural network, the image pixels will be assigned as the values for the first layer of nodes, and these nodes will *activate* their connected nodes of the next layer. There is a *weight* associated with each connection, and the value of each node at the next layer is a prespecified function of the weighted values of its connected nodes. Each layer of nodes is activated,

one by one, by the setting of the node values for the layer below. This procedure is called a *forward pass*.

There are two types of layers: those with weights to be trained and those with fixed functions (no weights to be trained). Common examples of the former are *fully connected* layers, in which the value of each node is a weighted sum of all the node values at the prior level, and *convolutional* layers, in which the value of each node is the result of applying a convolution function over a (typically small) subset of the node values.

A common way of training a neural network is to use a *stochastic gradient descent* (SGD) algorithm. For each training image, a forward pass is done to activate all nodes using the current weights. The values computed for each node are retained as intermediate states. At the last layer, an *error term* is calculated by comparing the predicted label probabilities with the true label. Then, the error terms are propagated back through the network with a *backward pass*. During the backward pass, the gradient of each connection weight is calculated from the error terms and the retained node values, and the connection weights (i.e., the model parameters) are updated using these gradients.

For efficiency, most training applications do each forward and backward pass with a batch of images (called a *mini-batch*) instead of just one image. For each image inside the mini-batch, there will be one set of node activation values during the forward pass and one set of error terms during the backward pass. Convolutional layers tend to have far fewer weights than fully connected layers, both because there are far fewer connections and because, by design, many connections share the same weight.

Video classification tasks often use a structure called a *recurrent neural network* (RNN). An RNN is made up of multiple layers with recurrent (i.e. feedback) connections, called recurrent layers, such that a static unrolling of the RNN would be a very deep network with shared weights between some of the layers. The *Long-Short Term Memory* (LSTM) layer [20] is one popular type of recurrent layers that is frequently used for vision and speech tasks to capture sequence information of the data [16, 29, 34]. The LSTM layer contains memory cells that "remember" knowledge of previous timestamps, and the memory is updated selectively at each timestamp, controlled by special gate functions. A common approach for using RNNs on vision tasks, such as video classification [16, 34] and image captioning [29], is to stack LSTM layers on top of CNN layers. The CNN layers serve as an encoder that converts each frame of a video into a feature vector and feeds the video as a sequence of feature vectors into the LSTM layers. In order to train the LSTM layers, a complete sequence of the image frames need to be in one mini-batch.

2.2 Deep learning using GPUs

GPUs are often used to train deep neural networks, because the primary computational steps match their single-instruction-multiple-data (SIMD) nature and they provide much more raw computing capability than traditional CPU cores. Most high end GPUs are on self-contained devices that can be inserted into a server machine, as illustrated in Figure 2. One key aspect of GPU devices is that they have dedicated local memory, which we will refer to as "GPU memory," and their computing elements are only efficient when working on data in that GPU memory. Data stored outside the device, in CPU memory, must first be brought into the GPU memory (e.g., via PCI DMA) for it to be accessed efficiently.
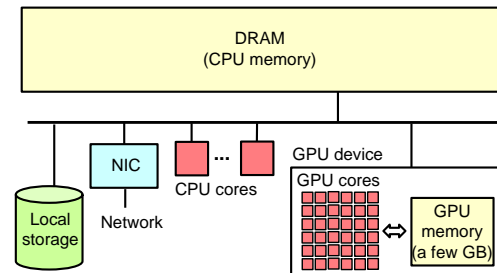

Figure 2. A machine with a GPU device.

Neural network training is an excellent match to the GPU computing model. For example, the forward pass of a fully connected layer, for which the value of each output node is calculated as the weighted sum of all input nodes, can be expressed as a matrix-matrix multiplication for a whole mini-batch. During the backward pass, the error terms and gradients can also be computed with similar matrix-matrix multiplications. These computations can be easily decomposed into SIMD operations and be performed efficiently with the GPU cores. Computations of other layers, such as convolution, have similar SIMD properties and are also efficient on GPUs. NVIDIA provides libraries for launching these computations on GPUs, such as the cuBLAS library [1] for basic linear algebra computations and the cuDNN library [2] for neural-network specific computations (e.g., convolution).

Caffe [21] is an open-source deep learning system that uses GPUs. In Caffe, a single-threaded worker launches and joins with GPU computations, by calling NVIDIA cuBLAS and cuDNN libraries, as well as some customized CUDA kernels. Each mini-batch of training data is read from an input file via the CPU, moved to GPU memory, and then processed as described above. For efficiency, Caffe keeps all model parameters and intermediate states in the GPU memory. As such, it is effective only for models and mini-batches small enough to be fully held in GPU memory. Figure 3 illustrates the CPU and GPU memory usage for a basic Caffe scenario.

2.3 Scaling ML with a parameter server

While early parallel ML implementations used direct message passing (e.g., via MPI) among threads for update exchanges, a *parameter server* architecture has become a popular approach to making it easier to build and scale ML applications
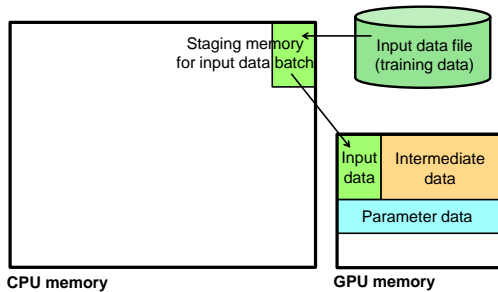
Figure 3. Single GPU ML, such as with default Caffe.

across CPU-based clusters [3, 4, 7, 10, 11, 14, 19, 25, 31, 36], particularly for data-parallel execution. Indeed, two of the largest efforts to address deep learning have used this architecture [7, 14].

Figure 4 illustrates the basic parameter server architecture. All state shared among application workers (i.e., the model parameters being learned) is kept in distributed shared memory implemented as a specialized key-value store called a "parameter server". An ML application's workers process their assigned input data and use simple `Read` and `Update` methods to fetch or apply a delta to parameter values, leaving the communication and consistency issues to the parameter server. The value type is often application defined, but is required to be serializable and be defined with an associative and commutative aggregation function, such as plus or multiply, so that updates from different workers can be applied in any order. In our image classification example, the value type could be an array of floating point values and the aggregation function could be plus.
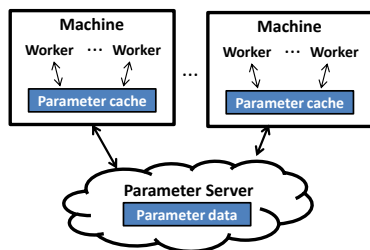

Figure 4. Parallel ML with parameter server.

To reduce remote communication, a parameter server system includes client-side caches that serve most operations locally. While some systems rely entirely on best-effort asynchronous propagation of parameter updates, many include an explicit `Clock` method to identify a point (e.g., the end of an iteration or mini-batch) at which a worker's cached updates should be pushed to the shared key-value store and its local cache state should be refreshed. The consistency model can conform to the Bulk Synchronous Parallel (BSP) model, in which all updates from the previous clock must be visible before proceeding to the next clock, or can use a looser but still bounded model. For example, the Stale Synchronous

Parallel (SSP) model [10, 19] allows the fastest worker to be ahead of the slowest worker by a bounded number of clocks. Both models have been shown to converge, experimentally and theoretically, with different tradeoffs.

While the picture illustrates the parameter server as separate from the machines executing worker threads, and some systems do work that way, the server-side parameter server state is commonly sharded across the same machines as the worker threads. The latter approach is particularly appropriate when considering a parameter server architecture for GPU-based ML execution, since the CPU cores and CPU memory is otherwise used only for input data buffering and preparation.

Given its proven value in CPU-based distributed ML, it is natural to use the same basic architecture and programming model with distributed ML on GPUs. To explore its effectiveness, we ported two applications (the Caffe system discussed above and a multi-class logistic regression (MLR) program) to a state-of-the-art parameter server system (IterStore [11]). Doing so was straightforward and immediately enabled distributed deep learning on GPUs, confirming the application programmability benefits of the data-parallel parameter server approach. Figure 5 illustrates what sits where in memory, to allow comparison to Figure 3 and designs described later.
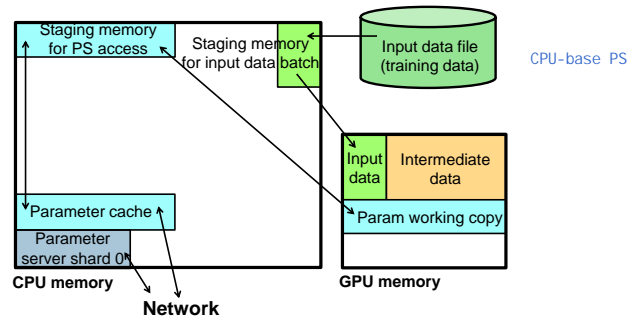

Figure 5. Distributed ML on GPUs using a CPU-based parameter server. The right side of the picture is much like the single-GPU illustration in Figure 3. But, a parameter server shard and client-side parameter cache are added to the CPU memory, and the parameter data originally only in the GPU memory is replaced in GPU memory by a local working copy of the parameter data. Parameter updates must be moved between CPU memory and GPU memory, in both directions, which requires an additional application-level staging area since the CPU-based parameter server is unaware of the separate memories.

While it was easy to get working, the performance was not acceptable. As noted by Chilimbi et al. [7], the GPU's computing structure makes it "extremely difficult to support data parallelism via a parameter server" using current implementations, because of GPU stalls, insufficient synchronization/consistency, or both. Also as noted by them and others [30, 33], the need to fit the full model, as well as a mini-batch of input data and intermediate neural network states, in the GPU mem-

ory limits the size of models that can be trained. The next section describes our design for overcoming these obstacles.

3. GPU-specialized parameter server design

This section describes three primary specializations to a parameter server to enable efficient support of parallel ML applications running on distributed GPUs: explicit use of GPU memory for the parameter cache, batch-based parameter access methods, and parameter server management of GPU memory on behalf of the application. The first two address performance, and the third expands the range of problem sizes that can be addressed with data-parallel execution on GPUs. Also discussed is the topic of execution model synchrony, which empirically involves a different choice for data-parallel GPU-based training than for CPU-based training.

3.1 Maintaining the parameter cache in GPU memory

One important change needed to improve parameter server performance for GPUs is to keep the parameter cache in GPU memory, as shown in Figure 6. (Section 3.3 discusses the case where everything does not fit.) Perhaps counter-intuitively, this change is not about reducing data movement between CPU memory and GPU memory—the updates from the local GPU must still be moved to CPU memory to be sent to other machines, and the updates from other machines must still be moved from CPU memory to GPU memory. Rather, moving the parameter cache into GPU memory enables the parameter server client library to perform these data movement steps in the background, overlapping them with GPU computing activity. Then, when the application uses the read or update functions, they proceed within the GPU memory. Putting the parameter cache in GPU memory also enables updating of the parameter cache state using GPU parallelism.
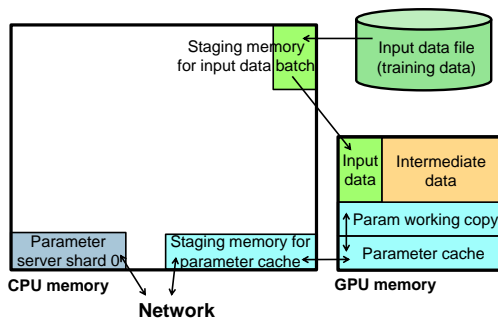

Figure 6. Parameter cache in GPU memory. In addition to the movement of the parameter cache box from CPU memory to GPU memory, this illustration differs from Figure 5 in that the associated staging memory is now inside the parameter server library. It is used for staging updates between the network and the parameter cache, rather than between the parameter cache and the GPU portion of the application.

3.2 Pre-built indexes and batch operations

Given the SIMD-style parallelism of GPU devices, per-value read and update operations of arbitrary model parameter values can significantly slow execution. In particular, performance problems arise from per-value locking, index lookups, and data movement. To realize sufficient performance, our GPU-specialized parameter server supports batch-based interfaces for reads and updates. Moreover, GeePS exploits the iterative nature of model training [11] to provide batch-wide optimizations, such as pre-built indexes for an entire batch that enable GPU-efficient parallel "gathering" and updating of the set of parameters accessed in a batch. These changes make parameter server accesses much more efficient for GPU-based training.

3.3 Managing limited GPU device memory

As noted earlier, the limited size of GPU device memory was viewed as a serious impediment to data-parallel CNN implementations, limiting the size of the model to what could fit in a single device memory. Our parameter server design addresses this problem by managing the GPU memory for the application and swapping the data that is not currently being used to CPU memory. It moves the data between GPU and CPU memory in the background, minimizing overhead by overlapping the transfers with the training computation, and our results demonstrate that the two do not interfere with one another.

Managing GPU memory inside the parameter server. Our GPU-specialized parameter server design provides read and update interfaces with parameter-server-managed buffers. When the application reads parameter data, the parameter server client library will *allocate* (user-level allocation implemented by the parameter server) a buffer in GPU memory for it and return the pointer to this buffer to the application, instead of copying the parameter data to an application-provided buffer. When the application finishes using the parameter data, it returns the buffer to the parameter server. We call those two interfaces `Read` and `PostRead`. When the application wants to update parameter data, it will first request a buffer from the parameter server using `PreUpdate` and use this buffer to store its updates. The application calls `Update` to pass that buffer back, and the parameter server library will apply the updates stored in the buffer and reclaim the buffer memory.

The application can also store their local non-parameter data (e.g., intermediate states) in the parameter server using similar interfaces. The local data will not be shared with the other application workers, so accessing the local data will be much faster than accessing the parameter data. For example, when the application reads the local data, the parameter server will just return a pointer that points to the stored local data, without copying it to a separate buffer. Similarly, the application can directly modify the requested local data, without needing to issue an explicit `Update` operation.

Method name	Input	Description	Blocking
`Read`	list of keys and data staleness bound	request a buffer, filled with parameter data	yes
`PostRead`	buffer from `Read` call	release the buffer	no
`PreUpdate`	list of keys	request an empty buffer, structured for parameter data	yes
`Update`	buffer from `PreUpdate` call	release the buffer and save the updates	no
`LocalAccess`	list of keys for local data	request a buffer, (by default) filled with local data	yes
`PostLocalAccess`	buffer from `LocalAccess` call	release the buffer and (by default) save the data in it	no
`TableClock`	table ID	commit all updates to one table	no

Table 1. GeePS API calls used for access to parameter data and GeePS-managed local data.

Swapping data to CPU memory when it does not fit. The parameter server client library will be able to manage all the GPU memory on a machine, if the application keeps all its local data in the parameter server and uses the PS-managed buffers. When the GPU memory of a machine is not big enough to host all data, the parameter server will store part of the data in the CPU memory. The application still accesses everything through GPU memory, as before, and the parameter server library will do the data movement for it. When the application `Reads` parameter data that is stored in CPU memory, the parameter server will perform this read using CPU cores and copy the data from CPU memory to an allocated GPU buffer, likewise for local data `Reads`. Figure 7 illustrates the resulting data layout in the GPU and CPU memories.
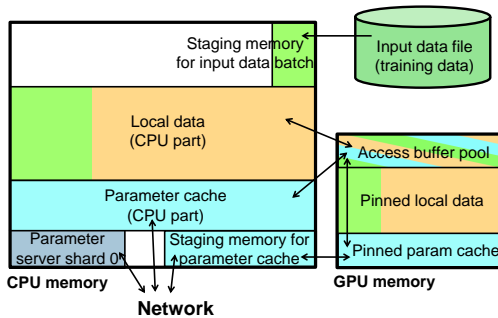

Figure 7. Parameter cache and local data partitioned across CPU and GPU memories. When all parameter and local data (input data and intermediate data) cannot fit within GPU memory, our parameter server can use CPU memory to hold the excess. Whatever amount fits can be pinned in GPU memory, while the remainder is transferred to and from buffers that the application can use, as needed.

GPU/CPU data movement in the background. Copying data between GPU and CPU memory could significantly slow down data access. To minimize slowdowns, our parameter server uses separate threads to perform the `Read` and `Update` operations in the background. For an `Update` operation, because the parameter server owns the update buffer, it can apply the updates in the background and reclaim the update buffer after it finishes. In order to perform the `Read` operations in the background, the parameter server will need to know in advance the sets of parameter data that the ap-

plication will access. Fortunately, iterative applications like neural network training typically apply the same parameter data accesses every iteration [11], so the parameter server can easily predict the `Read` operations and perform them in advance in the background.

3.4 Eschewing asynchrony

Many recent ML model training systems, including for neural network training, use a parameter server architecture to share state among data-parallel workers executing on CPUs. Consistent reports indicate that, in such an architecture, some degree of asynchrony (bounded or not) in parameter update exchanges among workers leads to significantly faster convergence than when using BSP [3, 7, 10, 14, 19, 24, 32]. We observe the opposite with data-parallel workers executing on GPUs—while synchronization delays can be largely eliminated, as expected, convergence is much slower with the more asynchronous models because of reduced training quality. This somewhat surprising observation is supported and discussed further in Section 5.4.

4. GeePS implementation

This section describes GeePS, a GPU-specialized parameter server system that implements the design aspects described in Section 3.

4.1 GeePS data model and API

GeePS is a C++ library that manages both the parameter data and local data for GPU-based machine learning applications (such as Caffe). The distributed application program usually creates one ML worker process on each machine and each of them links to one instance of the GeePS library. Algorithm 1 gives an example structure of a deep learning application using GeePS. The ML application worker often runs in a single CPU thread that launches NVIDIA library calls or customized CUDA kernels to perform computations on GPUs, and it calls GeePS functions to access and release GeePS-managed data. The GeePS APIs are summarized in Table 1.

GeePS manages all data as a collection of *rows* indexed by keys. The rows are then logically grouped into *tables*, and rows in the same table share the same attributes (e.g., data age). In our current implementation, each row is defined as a fixed sized array of float values, allowing efficient cross-machine communication without any marshalling. In our

deep learning application, because the model parameters (i.e., connection weights of each layer) can have different sizes, we store each model parameter as multiple rows in the same table.

GeePS implements the read and update operations with PS-managed buffers for parameter data access, and a pair of operations for local data access, with which the application can directly modify the accessed local data without an explicit update operation. GeePS also provides a `TableClock` operation for application workers to signal the completion of per-table updates, and the *data age* of a table (and the rows in it) is defined as the number of times that the `TableClock` operation is called on that table by all workers. Among all the API calls, `Read`, `PreUpdate`, and `LocalAccess` are blocking, forcing the application worker to wait when data or buffer space is not ready, and the other calls are all asynchronous and return immediately. By making the application worker wait on `Read`, GeePS supports three execution synchrony models: BSP, SSP [19], and Asynchrony.

Some of our specializations (pre-built indices, background `Read`, and data placement decisions) exploit knowledge of the operation sequence of the application. Previous work shows that one can easily get such operation sequence information from many iterative ML applications (including deep learning), because they do the same (or nearly the same) sequence of operations every iteration [11]. GeePS implements an operation sequence gathering mechanism like that described by Cui et al. [11]. It can gather the operation sequence either in the first iteration or in a *virtual iteration*. For example, in Algorithm 1, before the real training iterations start, the application performs a virtual iteration, with all GeePS calls being marked with a `virtual` flag, so that the operations are only recorded by GeePS but no real actions are taken. GeePS uses the gathered operation sequence knowledge as a hint to build the data structures, build the access indices, make GPU/CPU data placement decisions, and perform prefetching. Since the gathered access information is used only as a hint, knowing the exact operation sequence is not a requirement for correctness, but a performance optimization.[1]

4.2 GeePS architecture

Storing data. GeePS shards the parameter data across all instances, and each GeePS instance stores one shard of the parameter data in its *parameter server shard*. The parameter server shards are not replicated, and fault tolerance is handled by checkpointing. In order to reduce communication traffic, each instance has a *parameter cache* that stores a local

---

[1] For most DNN applications (including CNN and RNN), the application accesses all model parameters every mini-batch, so the gathered information is exact. For some applications with sparse training data (e.g., BOW representation for NLP tasks), the bottom layer of the network might just use a subset of the weights. Even for these tasks, the operation sequence of a whole epoch still repeats. The operation sequence only changes when the training data is shuffled across epochs, and, for this special case, we can choose to prefetch all the parameter data that can possibly be used, when there is enough memory.

Algorithm 1 A DNN application with GeePS

$L \leftarrow$ number of layers in the network
$paramDataKeys \leftarrow$ decide row keys for param data
$localDataKeys \leftarrow$ decide row keys for local data
 # Report access information with a virtual iteration
TRAINMINIBATCH($null$, $virtual = yes$)
 # Real training iterations
while not done **do**
    TRAINMINIBATCH($nextTrainData$, $virtual = false$)
end while
function TRAINMINIBATCH($trainData$, $virtual$)
    # Forward pass
    for $i = 0 \sim (L-1)$ **do**
        $paramDataPtr \leftarrow$
        $geeps.Read(paramDataKeys_i, virtual)$
        $localDataPtr \leftarrow$
        $geeps.LocalAccess(localDataKeys_i, virtual)$
        if not $virtual$ **then**
            Setup $layer_i$ with data pointers
            Forward computation of $layer_i$
        end if
        $geeps.PostRead(paramDataPtr)$
        $geeps.PostLocalAccess(localDataPtr)$
    end for
    # Backward pass
    for $i = (L-1) \sim 0$ **do**
        $paramDataPtr \leftarrow$
        $geeps.Read(paramDataKeys_i, virtual)$
        $paramUpdatePtr \leftarrow$
        $geeps.PreUpdate(paramDataKeys_i, virtual)$
        $localDataPtr \leftarrow$
        $geeps.LocalAccess(localDataKeys_i, virtual)$
        if not $virtual$ **then**
            Setup $layer_i$ with data pointers
            Backward computation of $layer_i$
        end if
        $geeps.PostRead(paramDataPtr)$
        $geeps.Update(paramUpdatePtr)$
        $geeps.PostLocalAccess(localDataPtr)$
        $geeps.TableClock(table = i, virtual)$
    end for
end function

---

snapshot of the parameter data, and the parameter cache is refreshed from the parameter server shards, such as at every clock for BSP. When the application applies updates to the parameter data, those updates are also stored in the parameter cache (a write-back cache) and will be submitted to the parameter server shards at the end of every clock (when a `TableClock` is called). The parameter cache has two parts, a GPU-pinned parameter cache and a CPU parameter cache. If everything fits in GPU memory, only the GPU parameter cache is used. But, if the GPU memory is not big enough, GeePS will keep some parameter data in the CPU parameter cache. (The data placement policies are described in Section 4.4.) Each GeePS instance also has an *access buffer pool* in GPU memory, and GeePS allocates GPU buffers for `Read` and `PreUpdate` operations from the buffer pool. When `PostRead` or `Update` operations are called, the memory will be reclaimed by the buffer pool. GeePS manages application's input data and intermediate states as *local data*. The local data also has a GPU-pinned part and a CPU part, with the

CPU part only used if necessary. GeePS divides the key space into multiple *partitions*, and the rows in different partitions are physically managed in different data structures and with different sets of communication threads.

Data movement across machines. GeePS performs communication across machines asynchronously with three types of background threads: *keeper* threads manage the parameter data in parameter server shards; *pusher* threads send parameter data updates from parameter caches to parameter server shards, by sending messages to keeper threads; *puller* threads receive parameter data from parameter server shards to parameter caches, by receiving messages from keeper threads. The communication is implemented using sockets, so the data needs to be copied to some CPU staging memory before being sent through the network, and the received data will also be in the CPU staging memory. The pusher/puller threads perform data movement between CPU memory and GPU memory using CUDA APIs.

Data movement inside a machine. GeePS uses two background threads to perform the data access operations for the application workers. The *allocator* thread performs the `Read`, `PreUpdate`, and `LocalAccess` operations by allocating buffers from the buffer pool and copying the requested data to the buffers. The *reclaimer* thread performs the `PostRead`, `Update`, and `PostLocalAccess` operations by saving the data to parameter cache or local store and reclaiming the buffers back to the buffer pool. These threads assign and update parameter data in large batches with pre-built indices by launching CUDA kernels on GPUs, as described in Section 4.3.

Synchronization and data freshness guarantees. GeePS supports BSP, asynchrony, and the Staleness Synchronous Parallel (SSP) model [19], wherein a worker at clock t is guaranteed to see all updates from all workers up to clock $t - 1 - slack$, where the slack parameter controls the data freshness. SSP with a slack of zero is the same as BSP.

To enforce SSP bounds, each parameter server shard keeps a vector clock for each table, where each vector clock entry stores the number of times each worker calls the `TableClock` operation on that table. The *data age* of each table in a parameter server shard is the minimal value of the corresponding vector clock. The parameter cache also keeps the data age information with the cached data, and the allocator thread is blocked when the data is not fresh enough.

Locking. GeePS's background threads synchronize with each other, as well as the application threads, using mutex locks and condition variables. Unlike some other CPU-based parameter servers that use per-row locks [10, 11, 32], we employ a coarse-grained locking design, where one set of mutex lock and condition variable is used for a whole key partition. We make this design decision for two reasons. First, with coarse-grained locking, batched data operations can be easily performed on a whole partition of rows. Second, unlike CPU applications, where one application thread is launched

for each CPU core, a GPU application often has just one CPU host thread interacting with each GeePS instance, making lock contention less of an issue.

4.3 Parallelizing batched access

GeePS provides a key-value store interface to the application, where each parameter row is named by a unique key. When the application issues a read or update operation (for accessing a set of model parameters), it will provide a list of keys for the target rows. GeePS could use a hash map to map the row keys to the locations where the rows are stored. But, in order to make the batched access be executed by all GPU cores, GeePS will use the following mechanism. Suppose the application update n rows, each with m floating point values, in one `Update` operation, it will provide an array of n parameter row updates $\{\{updates[i][j]\}_{j=1}^{m}\}_{i=1}^{n}$, and (provided in `PreUpdate`) an array of n keys $\{keys[i]\}_{i=1}^{n}$. GeePS will use an index with n entries, where each of $\{index[i]\}_{i=1}^{n}$ stores the location of the cached parameter update. Then, it will do the following data operation for this `Update`: $\{\{parameters[index[i]][j] \mathrel{+}= updates[i][j]\}_{j=1}^{m}\}_{i=1}^{n}$. This operation can be executed with all the GPU cores. Moreover, the index can be built just once for each batch of keys, based on the operation sequence gathered as described earlier, and re-used for each instance of the given batch access.

4.4 GPU memory management

GeePS keeps the GPU-pinned parameter cache, GPU-pinned local data, and access buffer pool in GPU memory. They will be all the GPU memory allocated in a machine if the application keeps all its input data and intermediate states in GeePS and uses the GeePS-managed buffers. GeePS will pin as much parameter data and local data in GPU memory as possible. But, if the GPU memory is not large enough, GeePS will keep some of the data in CPU memory (the CPU part of the parameter cache and/or CPU part of the local data).

In the extreme case, GeePS can keep all parameter data and local data in the CPU memory. But, it will still need the buffer pool to be in the GPU memory, and the buffer pool needs to be large enough to keep all the *actively used data* even at peak usage. We refer to this peak memory usage as *peak size*. In order to perform the GPU/CPU data movement in the background, GeePS does double buffering by making the buffer pool twice as large as the peak size.

Data placement policy. We will now describe our policy for choosing which data to pin in GPU memory. In our implementation, any local data that is pinned in GPU memory does not need to use any access buffer space. The allocator thread will just give the pointer to the pinned GPU local data to the application, without copying the data. For the parameter data, even though it is pinned in GPU memory, the allocator thread still needs to copy it from the parameter cache to an access buffer, because the parameter cache could be modified by the background communication thread (the

puller thread) while the application is doing computation. As a result, pinning local data in GPU memory gives us more benefit than pinning parameter cache data. Moreover, if we pin the local data that is used at the peak usage, we can reduce the peak access buffer usage, because it does not need the buffer, allowing us to reserve less memory for the access buffer.

Algorithm 2 GPU/CPU data placement policy

Input: $\{paramData\}$, $\{localData\}$ ← entries of all parameter data and local data accessed at each layer
Input: $totalMem$ ← the amount of GPU memory to use
 # Start with everything in CPU memory
$\{cpuMem\} \leftarrow \{paramData\} \cup \{localData\}$
$\{gpuMem\} \leftarrow \emptyset$
 # Set access buffer twice the peak size for double buffering
$peakSize \leftarrow$ peak data usage, excluding GPU local data
$bufferSize \leftarrow 2 \times peakSize$
$availMem \leftarrow totalMem - bufferSize$
 # First pin local data used at peak
while $\exists data \in \{localData\} \cap \{cpuMem\} \cap \{peakLayer\}$ **do**
    $peakSizeDelta \leftarrow$
        $peakSize$ change if $data$ is moved to $\{gpuMem\}$
    $memSizeDelta \leftarrow size(data) + 2 \times peakSizeDelta$
    if $availMem < memSizeDelta$ **then**
        break
    end if
    Move $data$ from $\{cpuMem\}$ to $\{gpuMem\}$
    $availMem \leftarrow availMem - memSizeDelta$
end while
 # Pin more local data using the available memory
for each $data \in \{localData\} \cap \{cpuMem\}$ **do**
    if $availMem \geq size(data)$ **then**
        Move $data$ from $\{cpuMem\}$ to $\{gpuMem\}$
        $availMem \leftarrow availMem - size(data)$
    end if
end for
 # Pin parameter data using the available memory
for each $data \in \{paramData\} \cap \{cpuMem\}$ **do**
    if $availMem \geq size(paramData)$ **then**
        Move $data$ from $\{cpuMem\}$ to $\{gpuMem\}$
        $availMem \leftarrow availMem - size(data)$
    end if
end for
 # Dedicate the remaining available memory to the access buffer
Increase $bufferSize$ by $availMem$

Algorithm 2 illustrates our GPU/CPU data placement policy, and it only runs at the setup stage, after the access information is gathered. The algorithm chooses the entries to pin in GPU memory based on the gathered access information and a given GPU memory budget. While keeping the access buffer pool twice the peak size for double buffering, our policy will first try to pin the local data that is used at the peak in GPU memory, in order to reduce the peak size and thus the size of the buffer pool. Then, it will try to use the available capacity to pin more local data and parameter cache data in GPU memory. Finally, it will add any remaining available GPU memory to the access buffer.

Avoiding unnecessary data movement. When the application accesses/post-accesses the local data that is stored in CPU memory, by default, the allocator/reclaimer thread will need to copy the data between the CPU memory and the allocated GPU memory. However, sometimes this data movement is not necessary. For example, when we train a deep neural network, the input data and intermediate data are overwritten every new mini-batch, and the old values from the last mini-batch can be safely thrown away. To avoid this unnecessary data movement, we allow the application to specify a `no-fetch` flag when calling `LocalAccess`, and it tells GeePS to just allocate an uninitialized piece of GPU memory, without fetching the data from CPU memory. Similarly, when the application calls `PostLocalAccess` with a `no-save` flag, GeePS will just free the GPU memory, without saving the data to CPU memory.

5. Evaluation

This section evaluates GeePS's support for parallel deep learning over distributed GPUs, using two recent image classification models and a video classification model executed in the original and modified Caffe application. The evaluation confirms four main findings: (1) GeePS provides effective data-parallel scaling of training throughput and training convergence rate, at least up to 16 machines with GPUs. (2) GeePS's efficiency is much higher, for GPU-based training, than a traditional CPU-based parameter server and also much higher than parallel CPU-based training performance reported in the literature. (3) GeePS's dynamic management of GPU memory allows data-parallel GPU-based training on models that are much larger than used in state-of-the-art deep learning for image classification and video classification. (4) For GPU-based training, unlike for CPU-based training, loose consistency models (e.g., SSP and asynchronous) significantly reduce convergence rate compared to BSP. Fortunately, GeePS's efficiency enables significant scaling benefits even with larger BSP-induced communication delays.

A specific non-goal of our evaluation is comparing the classification accuracies of the different models. Our focus is on enabling faster training of whichever model is being used, which is why we measure performance for several.

5.1 Experimental setup

Application setup. We use Caffe [21], the open-source single-GPU convolutional neural network application discussed earlier.[2] Our experiments use unmodified Caffe to represent the optimized single-GPU case and a minimally modified instance (GeePS-Caffe) that uses GeePS for data-parallel execution. GeePS-Caffe uses GeePS to manage all its parameter data and local data (including input data and intermediate data), using the same structure as illustrated in Algorithm 1. The parameter data of each layer is stored as rows of a distinct GeePS table, allowing GeePS to propagate

---

[2] For the image classification application, we used the version of Caffe from `https://github.com/BVLC/caffe` as of June 19, 2015. Since their master branch version does not support RNN, for the video classification application, we used the version from `https://github.com/LisaAnne/lisa-caffe-public/tree/lstm_video_deploy` as of Nov 9, 2015.

the each layer's updates during the computations of other layers, as suggested by Zhang et al. [35]. Each GeePS row is configured to be an array of 128 float values.

Cluster setup. Each machine in our cluster has one NVIDIA Tesla K20C GPU, which has 5 GB of GPU device memory. In addition to the GPU, each machine has four 2-die 2.1 GHz 16-core AMD® Opteron 6272 packages and 128 GB of RAM. Each machine is installed with 64-bit Ubuntu 14.04, CUDA toolkit 7.5, and cuDNN v2. The machines are inter-connected via a 40 Gbps Ethernet interface (12 Gbps measured via iperf), and Caffe reads the input training data from remote file servers via a separate 1 Gbps Ethernet interface.

Image classification datasets and models. The experiments use two datasets for image classification. The first one is the ImageNet22K dataset [15], which contains 14 million images labeled to 22,000 classes. Because of the computation work required to train such a large dataset, CPU-based systems running on this dataset typically need a hundred or more machines and spend over a week to reach convergence [7]. We use half of the images (7 million images) as the training set and the other half as the testing set, which is the same setup as described by Chilimbi et al. [7]. For the ImageNet22K dataset, we use a similar model to the one used to evaluate ProjectAdam [7], which we refer to as the *AdamLike* model.[3] The AdamLike model has five convolutional layers and three fully connected layers. It contains 2.4 billion connections for each image, and the model parameters are 470 MB in size.

The second dataset we used is Large Scale Visual Recognition Challenge 2012 (ILSVRC12) [26]. It is a subset of the ImageNet22K dataset, with 1.3 million images labeled to 1000 classes. For this dataset, we use the *GoogLeNet* model [28], a recent inception model from Google. The network has about 100 layers, and 22 of them have model parameters. Though the number of layers is large, the model parameters are only 57 MB in size, because they use mostly convolutional layers.

Video classification datasets and models. We use the UCF-101 dataset [27] for our video classification experiments. UCF-101 has about 8,000 training videos and 4,000 testing videos categorized into 101 human action classes. We use a recurrent neural network model for this application, following the approach described by Donahue et al. [16]. We use the GoogLeNet network as the CNN layers and stack LSTM layers with 256 hidden units on top of them. The weights of the GoogLeNet layers have been pre-trained with the single frames of the training videos. Following the same approach as is described by Donahue et al. [16], we extract the video frames at a rate of 30 frames per second and train the model with randomly selected video clips of 32 frames each.

---

[3] We were not able to obtain the exact model that ProjectAdam used, so we emulated it based on the descriptions in the paper. Our emulated model has the same number and types of layers and connections, and we believe our training performance evaluations are representative even if the resulting model accuracy may not be.

Training algorithm setup. Both the unmodified and the GeePS-hosted Caffe train the models using the SGD algorithm with a momentum of 0.9. Unless otherwise specified, we use the configurations listed in Table 2. Our experiments in Section 5.3 evaluate performance with different mini-batch sizes. For AdamLike and GoogLeNet network, we used the same learning rate for both single-machine training and distributed training, because we empirically found that this learning rate is the best setting for both. For the RNN model, we used a different learning rate for single-machine training, because it leads to faster convergence.

| Model	Mini-batch size (per machine)	Learning rate
AdamLike	200 images	0.0036, divided by 10 every 3 epochs
GoogLeNet	32 images	0.0036, divided by 10 every 150 epochs
RNN	1 video, 32 frames each	0.0000125 for 8 machines, 0.0001 for single-machine, divided by 10 every 20 epochs

Table 2. Model training configurations.

GeePS setup. Unless otherwise specified, we let GeePS keep the parameter cache and local data in GPU memory for our experiments, since it all fits for all of the models used; Section 5.3 evaluates performance when keeping part of the data in CPU memory, including for a very large model scenario. Unless otherwise specified, the BSP mode is used; Section 5.4 analyzes the effect of looser synchronization models.

5.2 Scaling deep learning with GeePS

This section evaluates how well GeePS supports data-parallel scaling of GPU-based training on both image classification and video classification application. We compare GeePS with three classes of systems: (1) *Single-GPU optimized training*: the original unmodified Caffe system (referred to as "Caffe") represents training optimized for execution on a single GPU. (2) *GPU workers with CPU-based parameter server*: multiple instances of the modified Caffe linked via IterStore [11] a state-of-the-art CPU-based parameter server ("CPU-PS"). (3) *CPU workers with CPU-based parameter server*: reported performance numbers from recent literature are used to put the GPU-based performance into context relative to state-of-the-art CPU-based deep learning.

Image classification. Figure 8 shows the training throughput scalability of the image classification application, in terms of both the number of images trained per second and the number of network connections trained per second. Note that there is a linear relationship between those two metrics. GeePS scales almost linearly when we add more machines. Compared to the single-machine optimized Caffe, GeePS achieves $13\times$ speedups on both GoogLeNet and AdamLike model with 16 machines. Compared to CPU-PS, GeePS achieves over $2\times$ more throughput. The GPU stall time of GeePS is
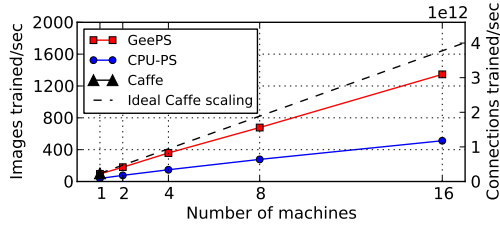
(a) AdamLike model on ImageNet22K dataset.
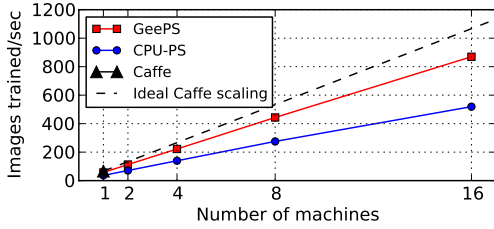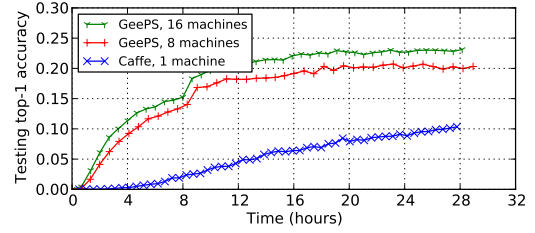

(b) GoogLeNet model on ILSVRC12 dataset.

Figure 8. Image classification throughput scalability. Both GeePS and CPU-PS run in the BSP mode.
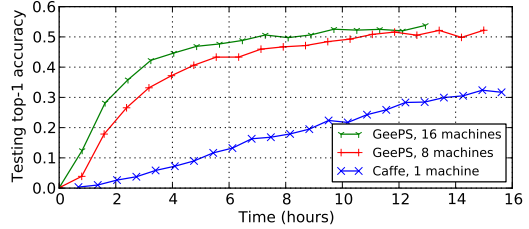

(a) AdamLike model on ImageNet22K dataset.

(b) GoogLeNet model on ILSVRC12 dataset.

Figure 9. Image classification top-1 accuracies.

only 8% for both GoogLeNet and AdamLike model, so 92% of the total runtime is spent on the application's computational work. While using CPU-PS, the GPU stall time is 51% and 65% respectively.

Chilimbi et al. [7] report that ProjectAdam trains 570 billion connections per second on the ImageNet22K dataset when with 108 machines (88 CPU-based worker machines with 20 parameter server machines) [7]. Figure 8(a) shows that GeePS achieves higher throughput with only 4 GPU machines, because of its efficient data-parallel execution on GPUs.

Figure 9 shows the image classification top-1 testing accuracies of the trained models. The top-1 classification accuracy is defined as the fraction of the testing images that are correctly classified. To evaluate convergence speed, we will compare the amount of time required to reach a given level of accuracy, which is a combination of image training throughput and model convergence per trained image. For the AdamLike model on the ImageNet22K dataset, Caffe needs 26.9 hours to reach 10% accuracy, while GeePS needs only 4.6 hours with 8 machines ($6\times$ speedup) or 3.3 hours with 16 machines ($8\times$ speedup). For the GoogLeNet model on the ILSVRC12 dataset, Caffe needs 13.7 hours to reach 30% accuracy, while GeePS needs only 2.8 hours with 8 machines ($5\times$ speedup) or 1.8 hours with 16 machines ($8\times$ speedup). The model training time speedups compared to the single-GPU optimized Caffe are lower than the image training throughput speedups, as expected, because each machine

determines gradients independently. Even using BSP, more training is needed than with a single worker to make the model converge. But, the speedups are still substantial.

For the AdamLike model on the ImageNet22K dataset, Chilimbi et al. [7] report that ProjectAdam needs one day to reach 13.6% accuracy with 58 machines (48 CPU-based worker machines with 10 parameter server machines). GeePS needs only 6 hours to reach the same accuracy with 16 machines (about $4\times$ speedup). To reach 13.6% accuracy, the DistBelief system trained (a different model) with 2,000 machines for a week [14].

Because both GeePS and CPU-PS run in the BSP mode, with the same number of machines, the accuracy improvement speedups of GeePS over CPU-PS are the same as the throughput speedups, so we leave them out of the graphs.

Video classification. Figure 10(a) shows the training throughput of the video classification application. GeePS scales linearly from Caffe ($8\times$ throughput with 8 machines). Figure 10(b) shows the top-1 testing accuracies. To reach 60% accuracy, Caffe used 3.6 hours, while GeePS with 8 machines used 0.5 hours ($7\times$ speedup); to reach 68% accuracy, Caffe used 8.4 hours, while GeePS with 8 machines used 2.4 hours ($3.5\times$ speedup).
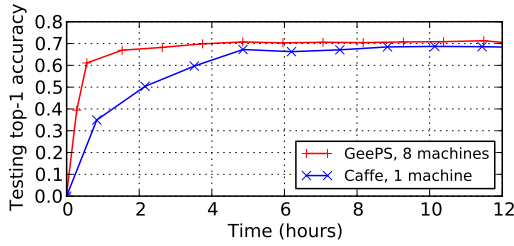
5.3 Dealing with limited GPU memory

An oft-mentioned concern with data-parallel deep learning on GPUs is that it can only be used when the entire model, as well as all intermediate state and the input mini-batch, fit
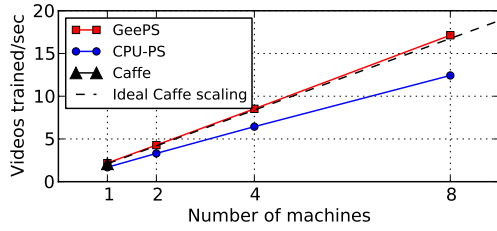
(a) Training throughput

(b) Top-1 accuracies.

Figure 10. Video classification scalability.

in GPU memory. GeePS eliminates this limitation with its support for managing GPU memory and using it to buffer data from the much larger CPU memory. Although all of the models we experiment with (and most state-of-the-art models) fit in our GPUs' 5 GB memories, we demonstrate the efficacy of GeePS's mechanisms in two ways: by using only a fraction of the GPU memory for the largest case (AdamLike) and by experimenting with a much larger synthetic model. We also show that GeePS's memory management support allows us to do video classification on longer videos.

Artificially shrinking available GPU memory. With a mini-batch of 200 images per machine, training the Adam-Like model on the ImageNet22K dataset requires only 3.67 GB memory per machine, with 123 MB for input data, 2.6 GB for intermediate states, and 474 MB each for parameter data and computed parameter updates. Note that the sizes of the parameter data and parameter updates are determined by the model, while the input data and intermediate states grow linearly with the mini-batch size. For best throughput, GeePS also requires use of an access buffer that is large enough to keep the actively used parameter data and parameter updates at the peak usage, which is 528 MB minimal and 1.06 GB for double buffering (the default) to maximize overlapping of data movement with computation. So, in order to keep everything in GPU memory, the GeePS-based training needs 4.73 GB of GPU memory.

Recall, however, that GeePS can manage GPU memory usage such that only the data needed for the layers being processed at a given point need to be in GPU memory.
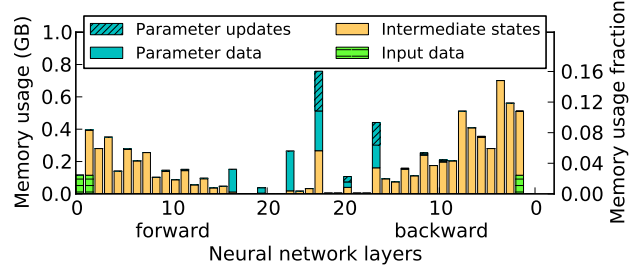

Figure 11. Per-layer memory usage of AdamLike model on ImageNet22K dataset.

Figure 11 shows the per-layer memory usage for training the AdamLike model, showing that it is consistently much smaller than the total memory usage. The left Y axis shows the absolute size (in GB) for a given layer, and the right Y axis shows the fraction of the absolute size over the total size of 4.73 GB. Each bar is partitioned into the sizes of input data, intermediate states, parameter data, and parameter updates for the given layer. Most layers have little or no parameter data, and most of the memory is consumed by the intermediate states for neuron activations and error terms. The layer that consumes the most memory uses about 17% of the total memory usage, meaning that about 35% of the 4.73 GB is needed for full double buffering.
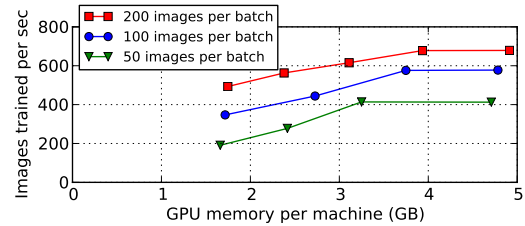

Figure 12. Throughput of AdamLike model on ImageNet22K dataset with different GPU memory budgets.

Figure 12 shows data-parallel training throughput using 8 machines, when we restrict GeePS to using different amounts of GPU memory to emulate GPUs with smaller memories. When there is not enough GPU memory to fit everything, GeePS must swap data to CPU memory. For the case of 200 images per batch, when we swap all data in CPU memory, we need only 35% of the GPU memory compared to keeping all data in GPU memory, but we are still able to get 73% of the throughput.

When the GPU memory limits the scale, people are often forced to use smaller mini-batch sizes to let everything fit in GPU memory. Our results in Figure 12 also shows that using our memory management mechanism is more efficient than shrinking the mini-batch size. For the three mini-batch sizes compared, we keep the inter-machine communication the same by doing multiple mini-batches per clock as needed

(e.g., four 50-image mini-batches per clock). For the case of 100 images per batch and 50 images per batch, 3.7 GB and 3.3 GB respectively are needed to keep everything in GPU memory (including access buffers for double buffering). While smaller mini-batches reduce the total memory requirement, they perform significantly less well for two primary reasons: (1) the GPU computation is more efficient with a larger mini-batch size, and (2) the time for reading and updating the parameter data locally, which does not shrink with mini-batch size, is amortized over more data.

Training a very large neural network. To evaluate performance for much larger neural networks, we create and train huge synthetic models. Each such neural network contains only fully connected layers with no weight sharing, so there is one model parameter (weight) for every connection. The model parameters of each layer is about 373 MB. We create multiple such layers and measure the throughput (in terms of # connections trained per second) of training different sized networks, as shown in Figure 13. For all sizes tested, up to a 20 GB model (56 layers) that requires over 70 GB total (including local data), GeePS is able to train the neural network without excessive overhead. The overall result is that GeePS's GPU memory management mechanisms allows data-parallel training of very large neural networks, bounded by the largest layer rather than the overall model size.
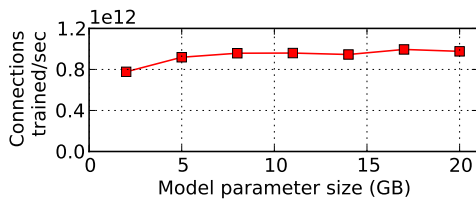

Figure 13. Training throughput on very large models. Note that the number of connections increases linearly with model size, so the per-image training time grows with model size because the per-connection training time stays relatively constant.

Video classification on longer videos. As is described in Section 2.1, the video classification application requires complete sequences of image frames to be in the same mini-batch, so the GPU memory size will either limit the maximum number of frames per video or force the model to be split across multiple machines, incurring extra complexity and network communication overhead. Using unmodified Caffe, for example, our RNN can support a maximum video length of 48 frames.[4] Because the videos are often sampled at a rate of 30 frames per second, a 48-frame video is less than 2 seconds in length. Ng et al. [34] find that using more frames in a video improves the classification accuracy. In order to use a video length of 120 frames, Ng et al. used a model-

---

[4] Here, we are just considering the memory consumption of the training stage. If we further consider the memory used for testing, the supported maximum video length will be shorter.

parallel approach to split the model across four machines, which incurs extra network communication overhead. By contrast, with the memory management support of GeePS, we are able to train videos with up to 192 frames, using solely data parallelism.

5.4 The effects of looser synchronization

Use of looser synchronization models, such as Stale Synchronous Parallel (SSP) or even unbounded asynchronous, has been shown to provide significantly faster convergence rates in data-parallel CPU-based model training systems [3, 7, 10, 14, 19, 24, 32]. This section shows results confirming our experience that this does not hold true with GPU-based deep learning.
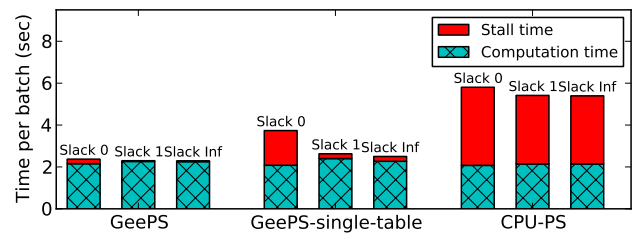

Figure 14. Data-parallel per-mini-batch training time for AdamLike model under different configurations. We refer to "stall time" as any part of the runtime when GPUs are not doing computations.

Figure 14 compares the per-mini-batch AdamLike model training time with 8 machines, when using GeePS, GeePS-single-table, and CPU-PS, and each of three synchronization models: BSP ("Slack 0"), SSP ("Slack 1"), and Asynchronous ("Slack Inf"). The GeePS-single-table setup has the application store all its parameter data in a single table, so that the parameter updates of all layers are sent to the server together as a whole batch. While in the default GeePS setup, where the parameter data of each layer is stored in a distinct table, the parameter updates of a layer can be sent to the server (and propagated to other workers) before the computation of other layers finish. Showing the performance of GeePS-single-table helps us understand the performance improvement coming from decoupling the parameter data of different layers. Each bar in Figure 14 divides the total into two parts. First, the computation time is the time that the application spends on training, which is mostly unaffected by the parameter server artifact and synchronization model; the non-BSP GeePS cases show a slight increase because of minor GPU-internal resource contention when there is great overlap between computation and background data movement. Second, the stall time includes any additional time spent on reading and updating parameter data, such as waiting time for slow workers to catch up or updated parameters to arrive, and time for moving data between GPU and CPU memory.

There are two interesting observations here. First, even for the Slack 0 (BSP) case, GeePS has little stall time. That is because, with our specializations, the per-layer updates can be propagated in the background, before the computations of other layers finish. The second observation is that expensive GPU/CPU data movement stalls CPU-PS execution, even with asynchronous communication (Slack Inf). Though CPU-PS also keeps parameter data of different layers in separate tables, the application has to perform expensive GPU/CPU data movement in the foreground each time it accesses the parameter data.
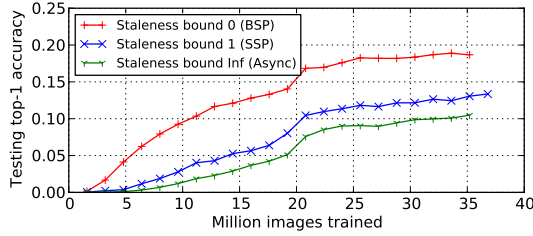

Figure 15. AdamLike model top-1 accuracy as a function of the number of training images processed, for BSP, SSP with slack 1, and Async.

Figure 15 compares the classification accuracy as a function of the number of training images processed, for GeePS with BSP, SSP (Slack 1), and Async. The result shows that, when using SSP (Slack 1) or Async, many more images must be processed to reach the same accuracy as with BSP (e.g., 2× more for Slack 1 and 3× more for Async to reach 10% accuracy). The training throughput speedups achieved with Slack 1 or Async are not sufficient to overcome the reduced training quality per image processed—using BSP leads to much faster convergence. We believe there are two reasons causing this outcome. First, with our specializations, there is little to no communication delay for DNN applications, so adding data staleness does not increase the throughput much. Second, the conditions in the SSP proofs of previous literatures [19] do not apply to DNN, because training a DNN is a non-convex problem. Interestingly, our observation is consistent with the concern about using parameter servers for data-parallel GPU-based training expressed by Chilimbi et al. [7]: "Either the GPU must constantly stall while waiting for model parameter updates or the models will likely diverge due to insufficient synchronization." Fortunately, GeePS's GPU-specialized design greatly reduces the former effect, allowing BSP-based execution to scale well.

6. Additional related work

This section augments the background related work discussed in Section 2, which covered use of parameter servers and individual GPUs for deep learning.

Coates et al. [9] describe a specialized multi-machine GPU-based system for parallel deep learning. The architecture used is very different than GeePS. Most notably, it relies on model parallelism to partition work across GPUs, rather than the simpler data-parallel model. It also uses specialized MPI-based communication over Infiniband, rather than a general parameter server architecture, regular sockets, and Ethernet.

Deep Image [33] is a custom-built supercomputer for deep learning via GPUs. The GPUs used have large memory capacity (12 GB), and their image classification network fits within it, allowing use of data-parallel execution. They also support for model-parallel execution, with ideas borrowed from Krizhevsky et al. [22], by partitioning the model on fully connected layers. The machines are interconnected by Infiniband with GPUDirect RDMA, so no CPU involvement is required, and they do not use the CPU cores or CPU memory to enhance scalability like GeePS does. Deep Image exploits its low latency GPU-direct networking for specialized parameter state exchanges rather than using a general parameter server architecture like GeePS.

MXNet [6] and Poseidon [35] are two concurrently developed ([12]) systems for multi-GPU deep learning. Both systems take the data-parallel approach by making use of CPU-based parameter servers (Poseidon uses Bosen [32] and MXNet uses ParameterServer [24]). GeePS differs from MXNet and Poseidon in two primary ways. First, in order to overcome the inefficiency of using CPU-based parameter servers, both MXNet and Poseidon rely on optimizations to their specific GPU application system (Poseidon is built on Caffe and MXNet writes their own). GeePS, on the other hand, specializes its reusable parameter server module, providing efficiency for all GPU deep learning applications it hosts. Indeed, the application improvements made for MXNet and Poseidon would further improve our reported performance numbers. Second, both MXNet and Poseidon require that each of their GPU machines has enough GPU memory to store all model parameters and intermediate states, limiting the size of their neural networks. GeePS's explicit GPU memory management support, on the other hand, allows the training of neural networks that are much bigger than the available GPU memory.

7. Conclusions

GeePS is a new parameter server for data-parallel deep learning on GPUs. Experimental results show that GeePS enables scalable training throughput, resulting in faster convergence of model parameters when using multiple GPUs and much faster convergence than CPU-based training. In addition, GeePS's explicit GPU memory management support enables GPU-based training of neural networks that are much larger than the GPU memory, swapping data to and from CPU memory in the background. Combined, GeePS enables use of data-parallel execution and the general-purpose parameter server model to achieve efficient, scalable deep learning on distributed GPUs.

Acknowledgments

References

[1] NVIDIA cuBLAS https://developer.nvidia.com/cublas.

[2] NVIDIA cuDNN https://developer.nvidia.com/cudnn.

[3] A. Ahmed, M. Aly, J. Gonzalez, S. Narayanamurthy, and A. J. Smola. Scalable inference in latent variable models. In *WSDM*, 2012.

[4] S. Ahn, B. Shahbaba, and M. Welling. Distributed stochastic gradient MCMC. In *ICML*, 2014.

[5] Y. Bengio, Y. LeCun, et al. Scaling learning algorithms towards AI. *Large-scale kernel machines*, 34(5), 2007.

[6] 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.

[7] T. Chilimbi, Y. Suzue, J. Apacible, and K. Kalyanaraman. Project Adam: Building an efficient and scalable deep learning training system. In *OSDI*, 2014.

[8] D. Ciresan, U. Meier, and J. Schmidhuber. Multi-column deep neural networks for image classification. In *CVPR*, 2012.

[9] A. Coates, B. Huval, T. Wang, D. Wu, B. Catanzaro, and N. Andrew. Deep learning with COTS HPC systems. In *ICML*, 2013.

[10] H. Cui, J. Cipar, Q. Ho, J. K. Kim, S. Lee, A. Kumar, J. Wei, W. Dai, G. R. Ganger, P. B. Gibbons, G. A. Gibson, and E. P. Xing. Exploiting bounded staleness to speed up big data analytics. In *USENIX ATC*, 2014.

[11] H. Cui, A. Tumanov, J. Wei, L. Xu, W. Dai, J. Haber-Kucharsky, Q. Ho, G. R. Ganger, P. B. Gibbons, G. A. Gibson, and E. P. Xing. Exploiting iterative-ness for parallel ML computations. In *SoCC*, 2014.

[12] H. Cui, G. R. Ganger, and P. B. Gibbons. Scalable deep learning on distributed GPUs with a GPU-specialized parameter server. CMU PDL Technical Report (CMU-PDL-15-107), 2015.

[13] G. E. Dahl, D. Yu, L. Deng, and A. Acero. Context-dependent pre-trained deep neural networks for large-vocabulary speech recognition. *IEEE Transactions on Audio, Speech, and Language Processing*, 20(1), 2012.

[14] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, A. Senior, P. Tucker, K. Yang, Q. V. Le, et al. Large scale distributed deep networks. In *NIPS*, 2012.

[15] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. ImageNet: A large-scale hierarchical image database. In *CVPR*, 2009.

[16] J. Donahue, L. A. Hendricks, S. Guadarrama, M. Rohrbach, S. Venugopalan, K. Saenko, and T. Darrell. Long-term recurrent convolutional networks for visual recognition and description. *arXiv preprint arXiv:1411.4389*, 2014.

[17] G. Gibson, G. Grider, A. Jacobson, and W. Lloyd. PRObE: A thousand-node experimental cluster for computer systems research. *USENIX ;login:*, 2013.

[18] G. Hinton, L. Deng, D. Yu, G. E. Dahl, A.-r. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. N. Sainath, et al. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal Processing Magazine*, 29(6), 2012.

[19] Q. Ho, J. Cipar, H. Cui, S. Lee, J. K. Kim, P. B. Gibbons, G. A. Gibson, G. R. Ganger, and E. P. Xing. More effective distributed ML via a Stale Synchronous Parallel parameter server. In *NIPS*, 2013.

[20] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural computation*, 9(8), 1997.

[21] 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.

[22] A. Krizhevsky. One weird trick for parallelizing convolutional neural networks. *arXiv preprint arXiv:1404.5997*, 2014.

[23] A. Krizhevsky, I. Sutskever, and G. E. Hinton. ImageNet classification with deep convolutional neural networks. In *NIPS*, 2012.

[24] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su. Scaling distributed machine learning with the parameter server. In *OSDI*, 2014.

[25] R. Power and J. Li. Piccolo: Building fast, distributed programs with partitioned tables. In *OSDI*, 2010.

[26] 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*, 2015.

[27] K. Soomro, A. R. Zamir, and M. Shah. Ucf101: A dataset of 101 human actions classes from videos in the wild. *arXiv preprint arXiv:1212.0402*, 2012.

[28] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. *arXiv preprint arXiv:1409.4842*, 2014.

[29] O. Vinyals, A. Toshev, S. Bengio, and D. Erhan. Show and tell: A neural image caption generator. *arXiv preprint arXiv:1411.4555*, 2014.

[30] M. Wang, T. Xiao, J. Li, J. Zhang, C. Hong, and Z. Zhang. Minerva: A scalable and highly efficient training platform for

deep learning. NIPS 2014 Workshop of Distributed Matrix Computations, 2014.

[31] Y. Wang, X. Zhao, Z. Sun, H. Yan, L. Wang, Z. Jin, L. Wang, Y. Gao, J. Zeng, Q. Yang, et al. Towards topic modeling for big data. *arXiv preprint arXiv:1405.4402*, 2014.

[32] J. Wei, W. Dai, A. Qiao, Q. Ho, H. Cui, G. R. Ganger, P. B. Gibbons, G. A. Gibson, and E. P. Xing. Managed communication and consistency for fast data-parallel iterative analytics. In *SoCC*, 2015.

[33] R. Wu, S. Yan, Y. Shan, Q. Dang, and G. Sun. Deep image: Scaling up image recognition. *arXiv preprint arXiv:1501.02876*, 2015.

[34] J. Yue-Hei Ng, M. Hausknecht, S. Vijayanarasimhan, O. Vinyals, R. Monga, and G. Toderici. Beyond short snippets: Deep networks for video classification. In *CVPR*, 2015.

[35] 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.

[36] R. Zhang and J. Kwok. Asynchronous distributed ADMM algorithm for global variable consensus optimization. In *ICML*, 2014.