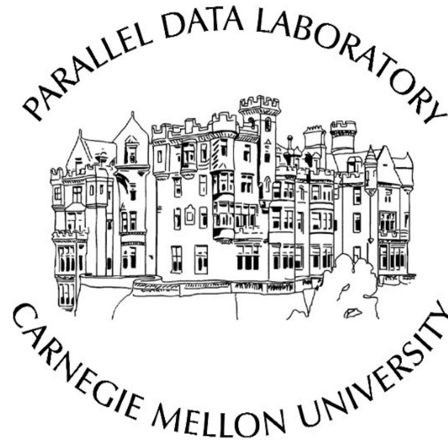


Scalable deep learning on distributed GPUs with a GPU-specialized parameter server

Henggang Cui, Gregory R. Ganger, and Phillip B. Gibbons
Carnegie Mellon University

CMU-PDL-15-107
October 2015

Parallel Data Laboratory
Carnegie Mellon University
Pittsburgh, PA 15213-3890

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 multiple GPUs was also considered unlikely to be effective, 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 9.5 times the number of training images processed per second on 16 machines (relative to the original optimized single-node code). Moreover, GeePS achieves the same training throughput with four GPU machines that a state-of-the-art CPU-only system achieves with 108 machines.

Acknowledgements: We thank the members and companies of the PDL Consortium (including Actifio, Avago, EMC, Facebook, Google, Hewlett-Packard Labs, Hitachi, Intel, Microsoft Research, MongoDB, NetApp, Oracle, Samsung, Seagate, Symantec, and Western Digital) for their interest, insights, feedback, and support. This research is supported in part by Intel as part of the Intel Science and Technology Center for Cloud Computing (ISTC-CC), National Science Foundation under awards CNS-1042537 and CNS-1042543 (PRObE [15]).

Keywords: Big Data infrastructure, Big Learning systems

1 Introduction

Large-scale deep learning is emerging as a primary machine learning approach for important, challenging problems such as image classification [13, 7, 20, 28] and speech recognition [12, 16]. 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 [13, 7]. Neural network training is known to map well to GPUs [8, 20], but many believe that this approach only works 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 [21, 10], GeePS handles the synchronization and communication complexities associated with sharing the model parameters being learned (in this case, the weights on the connections) across parallel workers. Unlike such previous systems, it 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 disperse parameters in the GPU, along with GPU-friendly caching, data staging, and memory management techniques.

GeePS supports a data-parallel implementation, in which the input data is partitioned among workers on different machines who 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 [28]. 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, 13, 9, 28, 19, 25], 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 [18], 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 $9.5\times$ using 16 machines. (The self-relative speed-up for GeePS is $14\times$, indicating near-linear speed-up.) In terms of classification accuracy, GeePS's rate of improvement on 16 machines is $4.9\times$ faster than the single-GPU optimized Caffe's. The training throughput achieved with just four GPU machines matches 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 $3\times$ faster than what was reported for a 58-machine ProjectAdam configuration. Interestingly, in contrast with recent work [17, 21, 7], 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 data staleness on accuracy improvements far outweighs the positive benefit 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 constantly moving data between CPU memory and GPU memory in the background, GeePS is able to keep the GPU engines busy without suffering 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 convolutional neural network [6], which consists of layered network of nodes and edges (connections), as depicted in Figure 1.

Since 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, 20, 13, 24]. The goal is to train a neural network to classify images (raw pixel maps) into pre-defined labels, using a set of training images with known labels. So, the first layer of the nodes (input of the network) are the 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.

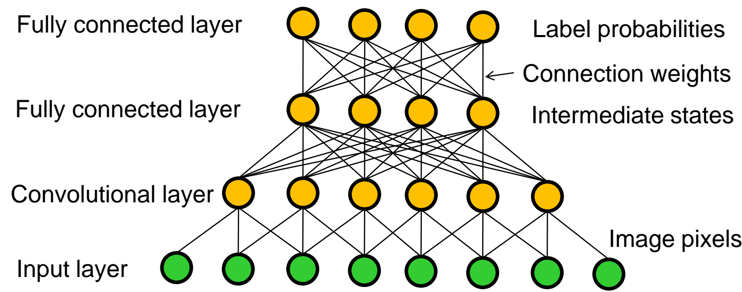


Figure 1: A convolutional neural network, with one convolutional layer and two fully connected layers.

A common way of training a neural network is to use *stochastic gradient descent* (SGD). 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. Moreover, 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.

2.2 Deep learning using GPUs

GPUs are often used to train deep neural networks, because the primary computational steps match their SIMD-style nature and they provide much more raw computing capability than traditional CPU cores. Most high end GPUs are on self-contained GPU 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.

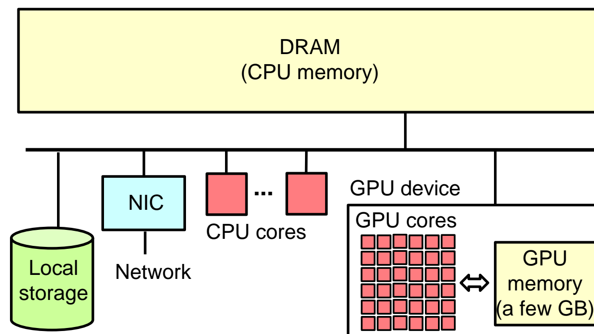


Figure 2: A machine with a GPU device.

Neural network training is an excellent match to the GPU computing model. During the forward pass, the value of each node in a fully connected layer is the weighted sum of the previous layer's node values, which can be calculated as a matrix-matrix multiplication for a given mini-batch and layer. Similarly, the value of each node in a convolutional layer involves the weighted sum of its connected node values. During the backward pass, the error terms and gradients are also calculated using similar matrix-matrix multiplications for fully connected layers, and variations for other layer types. These matrix-matrix multiplications, convolutions, and related operations can be easily decomposed to SIMD operations that can be performed by the GPU cores. NVIDIA provides a cuBLAS library [1], which can be used to launch basic linear algebra GPU computations from CPU code, and a cuDNN library [2] for launching GPU computations more specific to neural networks, such as calculating convolutions.

Caffe [18] 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.

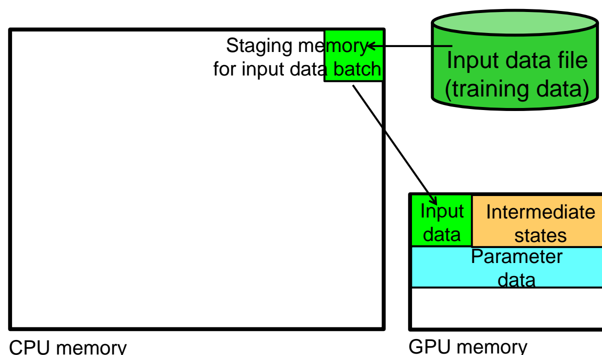


Figure 3: Single GPU ML, such as with default Caffe.

2.3 Scaling ML with a parameter server

While early parallel ML implementations used direct message passing among threads for update exchanges, a *parameter server* architecture has become a popular approach to making it easier to build and scale ML applications across CPU-based clusters [4, 22, 17, 7, 29, 26, 13, 5, 10, 11], particularly for data-parallel execution. Indeed, two of the largest recently-reported efforts to address deep learning have used this architecture [13, 7].

Figure 4 illustrates the basic parameter server architecture. All state shared among worker threads (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 worker threads process their assigned input data and use simple `Read-param` and `Update-param` 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 must be serializable and be defined with an associative and commutative aggregation function,

such as plus or multiply, so that updates from different worker threads 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.

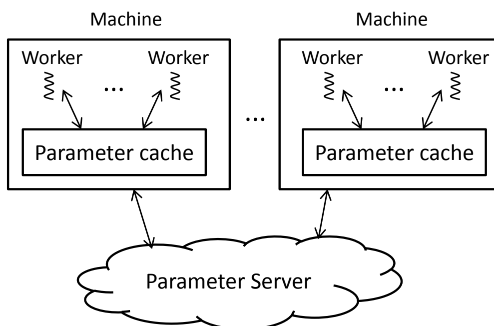


Figure 4: Parallel ML with parameter server.

To avoid constant 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 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 recently proposed SSP model [17, 10] 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 would be largely unused by the GPU-based workers.

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 program) to IterStore [11], a state-of-the-art parameter server system. 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.

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 [25, 28], the need to fit the full model, as well as a mini-batch of input data and intermediate neural network states, in the GPU memory limits the size of models that can be trained. The next section describes our design for overcoming these obstacles.

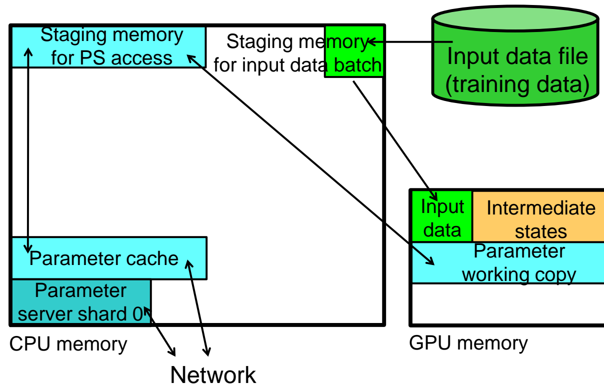


Figure 5: Distributed ML on GPUs using a CPU-based parameter server. The right side of the picture is much like the single-GPU illustration in Figure 3. But, a parameter server shard and client-side parameter cache are added to the CPU memory, and the parameter data originally only in the GPU memory 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.

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 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 problems 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.

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 locking, index lookups, and one-by-one data movement. To realize sufficient performance, our GPU-specialized parameter server supports batch-based interfaces for reads (`READ-BATCH`) and updates (`UPDATE-BATCH`).

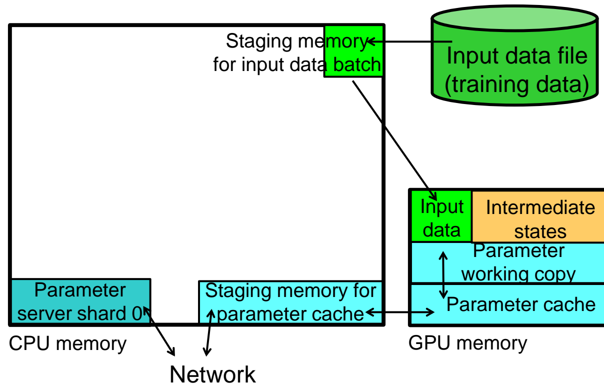


Figure 6: Parameter cache in GPU memory. In addition to the movement of the parameter cache box from CPU memory to GPU memory, this illustration differs from Figure 5 in that the associated staging memory is now inside the parameter server library. It is used for staging updates between the network and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

Moreover, GeePS exploits the repeating nature of iterative 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 servers 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 can move 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* a buffer in GPU memory for it and return the pointer to this buffer to the application, instead of copying the parameter data to a buffer provided by the application. When the application finishes using the parameter data, it returns the buffer to the parameter server. We call those two interfaces `BUFFER-READ-BATCH` and `POST-BUFFER-READ-BATCH`. When the application wants to update parameter data, it will first request a buffer from the parameter server using `PRE-BUFFER-UPDATE-BATCH` and use this buffer to store its updates. The application calls `BUFFER-UPDATE-BATCH` to pass that buffer back, and the parameter server library will apply the updates stored in the buffer and reclaim the buffer memory. To make it concise, in the rest of this paper, we will refer to the batched interfaces using PS-managed buffers as `READ`, `POST-READ`, `PRE-UPDATE`, and `UPDATE`. The application can also store their local non-parameter data (e.g., intermediate states) in the parameter server using the same interfaces, but with a `LOCAL` flag. 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 `UPDATE` operation.

Swapping data to CPU memory when it does not fit. By storing the local data in the parameter server, almost all GPU memory can be managed by the parameter server client library. 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 `READ`s parameter data that is stored in CPU memory, the parameter server will perform this read using a CPU core and copy the data from CPU memory to an allocated GPU buffer. Likewise, when the application `READ`s local data that is stored in CPU memory, the parameter server will copy the local data from CPU memory to an allocated GPU buffer. Figure 7 illustrates the resulting data layout in the GPU and CPU memories.

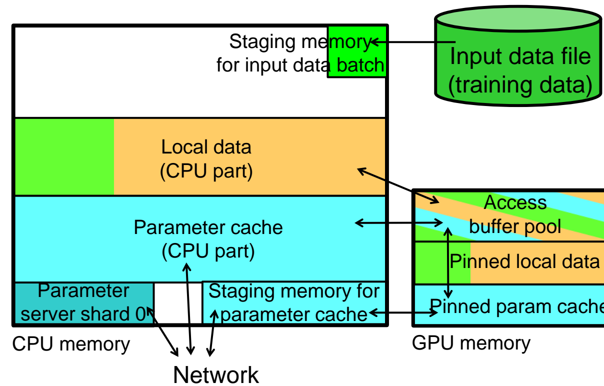


Figure 7: Parameter cache and local data partitioned across CPU and GPU memories. When all parameter and local data (input data and intermediate states) cannot fit within GPU memory, our parameter server can use CPU memory to hold the excess. Whatever amount fits can be pinned in GPU memory, while the 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 library will use 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 application will access. Fortunately, iterative applications like neural network training apply the same sequence of parameter server operations every clock [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 [17, 21, 7, 13, 10, 4, 27]. 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 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 machine learning applications. 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. For GPU-based ML applications (such as Caffe), the worker often runs in a single CPU thread and launches and joins with GPU computations, which might be NVIDIA library calls or customized CUDA kernels. On initializing the GeePS library, the application will provide the list of hosts that the program is running on, and each GeePS instance will create connections to the other hosts.

GeePS manages data as a collection of *rows* indexed by keys, and it implements the read and update operations with parameter-server-managed buffers, and we call them `READ`, `POST-READ`, `PRE-UPDATE`, and `UPDATE` for short. GeePS also provides a `CLOCK` function that allows the application to synchronize the workers. It supports three execution synchrony models: BSP, SSP [17], and asynchronous.

4.2 GeePS architecture

Storing data. Each GeePS instance stores one shard of the master version of the parameter data in its *parameter server shard*. The parameter server shard is 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 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. 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 `PRE-UPDATE` operations from the buffer pool. When `POST-READ` 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*, which is local to each worker. The local data also has a GPU-pinned part and a CPU part, with the CPU part only used if necessary.

Data movement across machines. GeePS performs communication across machines asynchronously with three types of communication threads: *keeper threads* manage the parameter data in parameter server shards; *pusher threads* move parameter data from parameter caches to parameter server shards, by sending messages to keeper threads; *puller threads* move parameter data from parameter server shards to parameter caches, by receiving messages from keeper threads. GeePS divides the key space into multiple partitions and manages the rows of different partitions in separate data structures with different sets of communication 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.

To send/receive data from/to GPU parameter cache, the pusher/puller threads will move the data between CPU memory and GPU memory.

Data movement inside a machine. GeePS uses two background GPU threads to perform the `READ` and `UPDATE` operations for the application. The *allocator* thread performs the `READ` and `PRE-UPDATE` operations by allocating buffers from the buffer pool and (only for `READ`) copying parameter values from the parameter cache to the buffers. The *reclaimer* thread performs the `POST-READ` and `UPDATE` operations by (only for `UPDATE`) applying updates from the buffers to the parameter cache and reclaiming the buffers back to the buffer pool. These threads assign and update parameter data in large batches with pre-built indices using GPUs, as described in Section 4.3.

Synchronization and data freshness guarantees. GeePS supports BSP, asynchrony, and the Staleness Synchronous Parallel (SSP) model [17], 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, where each entry stores the number of times each worker calls the `CLOCK` function. The *data age* of a parameter server shard is the minimal value of the vector clock. When the puller thread requests data from a parameter server shard, it will specify a data age requirement, and the keeper thread only sends the data when it is fresh enough. The parameter cache also keeps the data age information, and the allocator thread that performs the read at clock t will block if the age is less than $t - 1 - slack$, where the $slack$ is specified by the application.

Operation sequence gathering Some of our designs (pre-built indices, background `READ`, and data placement decisions) can benefit from knowing 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 sequence of operations every clock [11]. GeePS implements an operation sequence gathering mechanism like that described by Cui et al. [11]. It can gather the operation sequence information either in the first iteration or in a *virtual iteration*, during which the application just reports its sequence of operations without doing any real computation or keeping any states. GeePS uses the gathered operation sequence information to build the data structures (parameter server shard, parameter cache, and local data), build the access indices, prefetch the data (including cross-machine data fetching and background `READ`), and make GPU/CPU data placement decisions.

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 batched read or update operation, 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 keys $\{keys[i]\}_{i=1}^{n}$ and an array of n parameter row updates $\{\{updates[i][j]\}_{j=1}^{m}\}_{i=1}^{n}$. GeePS will use an index with n entries, where $\{index[i]\}_{i=1}^{n}$ is the location of the parameter data for $\{keys[i]\}_{i=1}^{n}$. 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 above, 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 stores 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 store 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.

Algorithm 1 GPU/CPU data placement policy

Start with everything in CPU memory
$total_mem \leftarrow$ the amount of GPU memory to use
$peak_size \leftarrow$ peak size of the actively used data
 # Twice the peak size for double buffering
if $total_mem < 2 \times peak_size$ **then**
    Throw an exception
end if
$used_mem \leftarrow 2 \times peak_size$
 # Try to reduce $peak_size$
while enough memory to for local data used at peak **do**
    Pin local data used at peak in GPU memory
    Update $used_mem$ with local data size
    Update $peak_size$
    Update $used_mem$ with the new $peak_size$
end while
 # Pin more local data using the available memory
while enough GPU memory and more local data **do**
    Pin more local data in GPU memory
    Update $used_mem$ with local data size
end while
 # Pin parameter data using the available memory
while enough GPU memory and more param data **do**
    Pin more param data in GPU memory
    Update $used_mem$ with param data size
end while

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. Moreover, if we pin the local data that is used at the peak usage, we can reduce the peak size, so that we can reserve less memory for the access buffer pool.
Algorithm 1 illustrates our GPU/CPU data placement policy. It chooses the entries to pin in GPU memory based on the gathered operation sequence information and a given GPU memory budget. We always keep 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 in GPU memory. Finally, it will use the rest available GPU memory to pin parameter cache data in GPU memory.

5 Evaluation

This section evaluates GeePS's support for parallel deep learning over distributed GPUs, using three recent image classification models 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. (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 image 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 [18], the open-source singe-GPU convolutional neural network application discussed earlier.[1] Our experiments use unmodified Caffe to represent the optimized single-GPU case and a minimally modified instance that uses GeePS for data-parallel execution.
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 and CUDA toolkit 6.5. 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.
Our experimental results and reported experiences are the result of 1000s of hours of machine time on a shared >$200K cluster. During execution of an experiment, no other activity uses the machines. Due to allocation rules and heavy contention for the cluster, each experiment was limited to 16 hours and usually to less than 10 machines.
Datasets and neural network models. The experiments use two image classification datasets. Dataset #1 is the ImageNet22K dataset [14], 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

[1] We used the version of Caffe from `https://github.com/BVLC/caffe` as of June 19, 2015.

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]. Dataset #2 is the Large Scale Visual Recognition Challenge 2012 (Ilsvrc12) dataset [23]. It is a subset of the ImageNet22K dataset, with 1.3 million images labeled to 1000 classes.

For the Ilsvrc12 dataset, we use two different models. First, the *AlexNet* model [20] is a popular neural network for image classification, and many deep learning systems borrow ideas from it [18, 7, 28]. It has five convolutional layers and three fully connected layers. Each convolutional layer is followed by a rectified linear unit (ReLU) layer, a max-pooling layer, and a local response normalization (LRN) layer. Each fully connected layer is followed by a rectified linear unit (ReLU) layer and a drop-out layer. There are 24 layers in total. Only the convolutional layers and fully connected layers have model parameters to be trained, and the other layers are fixed. The network has 0.7 billion connections for each image, and the model size is 244 MB. The number of model parameters is less than 0.7 billion because of weight sharing in convolutional layers.

The second model we use for the Ilsvrc12 dataset is the *GoogLeNet* model [24], 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.

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.[2] The AdamLike model has the same structure as the AlexNet model, with five convolutional layers and three fully connected layers, but each layer is larger. It contains 2.4 billion connections for each image, and the model parameters are 470 MB in size.

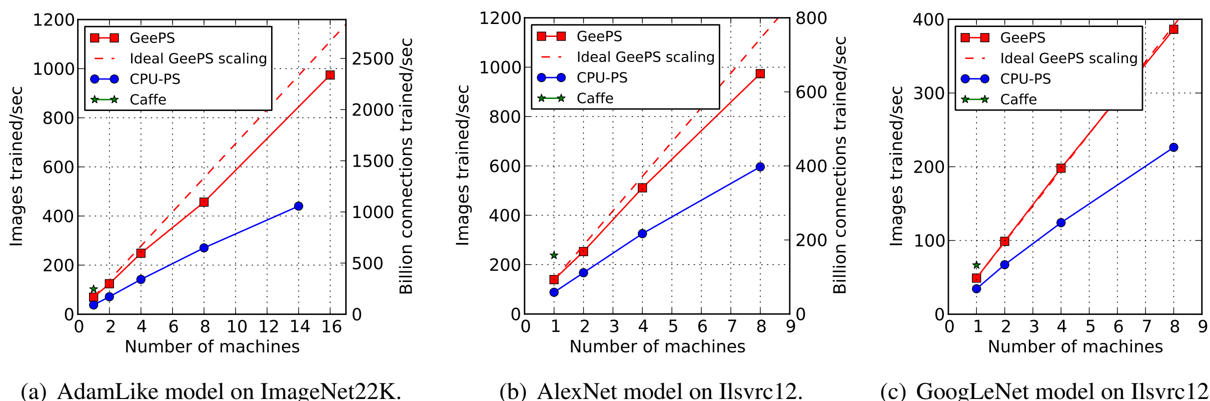
GeePS setup. We run one application worker (Caffe linked with one GeePS instance) on each machine. Unless otherwise specified, we let GeePS keep the parameter cache and local data in GPU memory for each experiments, since it all fits for all of the models used; Section 5.3 analyzes performance when keeping part of the data in CPU memory, including for a very large model scenario. Unless otherwise specified, BSP mode is used; Section 5.4 analyzes the effect of looser synchronization models.

5.2 Scaling image classification with GeePS

This section evaluates how well GeePS supports data-parallel scaling of GPU-based training. 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 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.

Figure 8 shows model training throughput, in terms of both number of images trained per second and number of neural network connections trained per second. Note that there is a linear relationship between those two metrics. Compared to Caffe, GeePS introduces overhead for explicit model parameter reads and updates, reducing training throughput by 26% (GoogLeNet) to 40% (AlexNet) when using just a single machine. But, GeePS scales almost linearly when we add more machines, achieving from 6.6× (AdamLike) to 7.9× (GoogLeNet) its single-machine throughput on 8 machines. On 16 machines, GeePS achieves 14× its single-machine throughput. The GoogLeNet scalability is so good because it uses many fewer model parameters, despite its predictive power. For all three benchmarks, the 8-machine throughput is over 4.5×

[2]We were not able to obtain the exact model that ProjectAdam uses, 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.

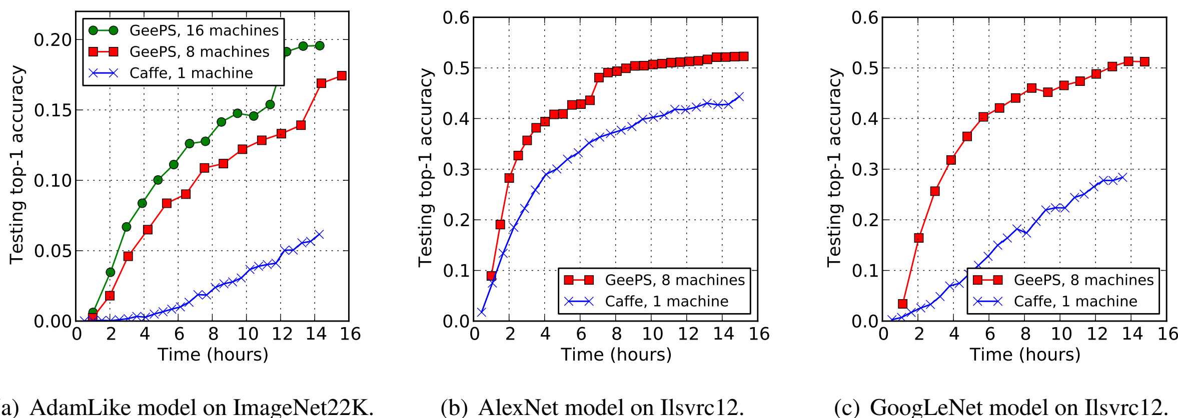


(a) AdamLike model on ImageNet22K. (b) AlexNet model on Ilsvrc12. (c) GoogLeNet model on Ilsvrc12.

Figure 8: Throughput scalability of three models on two datasets. Note that X-axis goes to 16 machines in (a).

that of the single-machine optimized Caffe. CPU-PS performs much less well, with larger single-machine overhead and less throughput scalability.

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



(a) AdamLike model on ImageNet22K. (b) AlexNet model on Ilsvrc12. (c) GoogLeNet model on Ilsvrc12.

Figure 9: Top-1 accuracies of three models on two datasets.

Figure 9 shows the top-1 image classification accuracies of our 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 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 12.2 hours to reach 5% accuracy, while GeePS needs only 3.3 hours using 8 machines (3.7x speedup). For the AlexNet model on the Ilsvrc12 dataset, Caffe needs 12.2 hours to reach 42% accuracy, while GeePS needs only 5.3 hours using 8 machines (2.3x speedup). For the GoogLeNet model on the Ilsvrc12 dataset, Caffe needs 13.2 hours to reach 28% accuracy, while GeePS needs only 3.3 hours using 8 machines (4.0x 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 using 58 machines (48 CPU-based worker machines with 10 parameter server machines). GeePS needs only 12.6 hours to reach the same accuracy using 8 machines, and 8.2 hours using 16 machines. To reach 13.6% accuracy, the DistBelief system trained (a different model) using 2,000 machines for a week [13].
Because both GeePS and CPU-PS run in the BSP mode using the same number of machines, the speedups of GeePS over CPU-PS are the same as the throughput speedups, so we leave them out of the graphs.

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 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' 5GB 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.
Artificially shrinking available GPU memory. With a mini-batch of 200 images per machine, training the AdamLike model on the ImageNet22K dataset requires only 3.67 GB, 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.

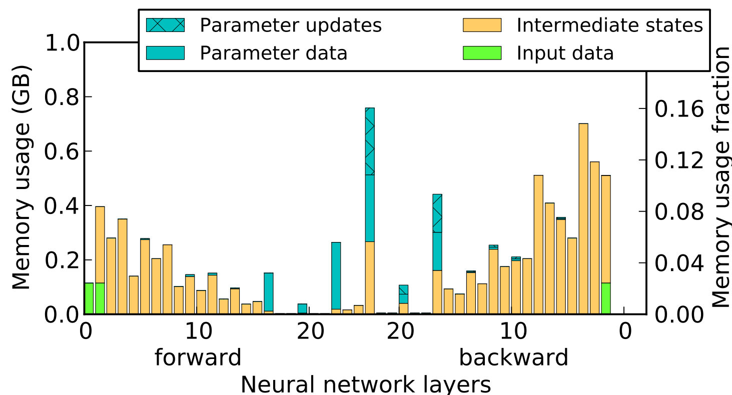


Figure 10: Per-layer memory usage of AdamLike model on ImageNet22K dataset.

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 10 shows the per-layer memory usage for

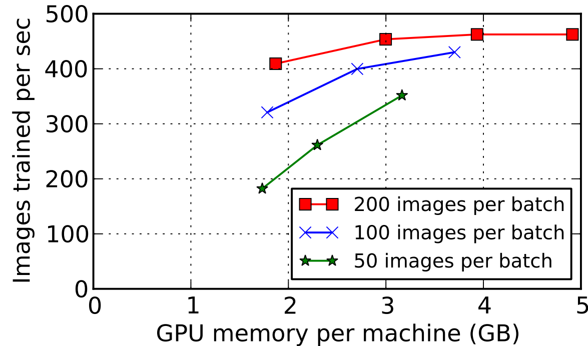


Figure 11: Throughput of AdamLike model on ImageNet22K dataset with different GPU memory budgets.

the AdamLike model training, 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 11 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 38%[3] of the GPU memory compared to keeping all data in GPU memory, but we are still able to get 88% of the throughput.

The same figure shows training throughput when using different mini-batch sizes, while keeping the inter-machine communication the same by doing multiple mini-batches per clock as needed (e.g., four 50-image mini-batches per clock). Because the sizes of input data and intermediate states grow linearly with the mini-batch size, the maximal memory needed for each case is different. For each curve, the leftmost point stands for the case where everything is kept in CPU memory, and we allocate enough access buffer for full double buffering. While smaller mini-batches reduce the total memory requirement significantly, 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. The 200-image-per-mini-batch case is much flatter because the aggregate parameter access overhead is smaller and the data movement between CPU and GPU memory is effectively overlapped with computation.

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 per second) of training different sized networks. Figure 12 shows the results. 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

[3]The fraction is larger than 35%, because there is some little amount of other memory usage in Caffe that is not input data, intermediate states, parameter data, or parameter updates.

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.

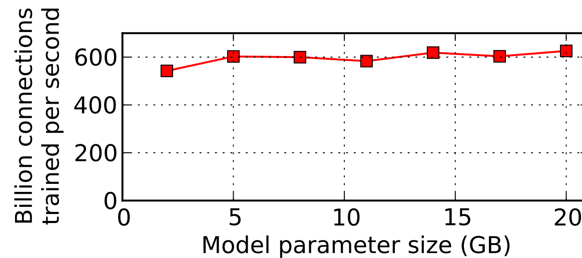


Figure 12: Training throughput on very large models. Note that the number of connections increases linearly with model size, so the per-image training time grows with model size because the per-connection training time stays relatively constant.

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 [17, 21, 7, 13, 10, 4, 27]. This section shows results confirming our experience that this does not hold true with GPU-based deep learning. While training throughput increases, because of greater ability to overlap communication with computation, it is not enough to overcome the effects of increased staleness in the shared parameters.

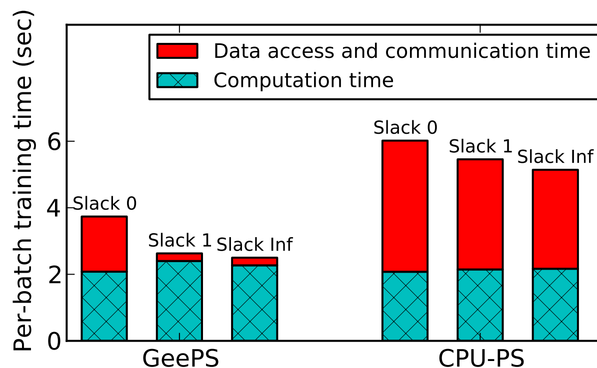


Figure 13: Data-parallel per-mini-batch training time for AdamLike model under different configurations.

Figure 13 compares the per-mini-batch AdamLike model training time when using GeePS and CPU-PS with 8 machines and each of three synchronization models: BSP ("Slack 0"), SSP ("Slack 1"), and Asynchronous ("Slack Inf"). Each bar divides the total into two parts. First, the computation time is the application training time, which is mostly unaffected by the 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 data access and communication time is the time

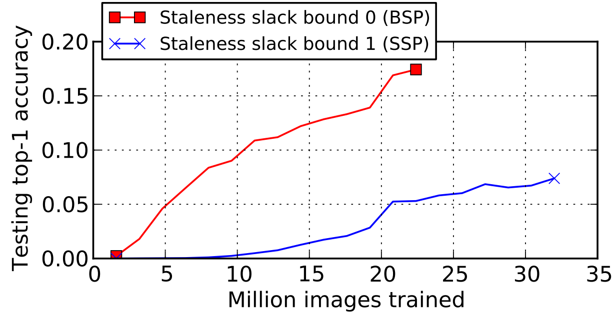


Figure 14: AdamLike model top-1 accuracy as a function of the number of training images processed, for BSP and SSP with slack 1.

spent on reading and updating parameter data, including any waiting for slow workers to catch up or updated parameters to arrive. As expected, the results show that when Slack 0 (BSP) involves the most data access and communication time, because of the need to copy parameter updates between CPU and GPU memory and exchange the updates over the network. With Slack 1, which allows a faster worker to be one clock ahead of the slowest worker, GeePS eliminates most data access and communication time and is over 2× faster than CPU-PS. With slack, GeePS can complete almost all data access and communication in the background, while the expensive GPU/CPU data movement stalls CPU-PS execution even with asynchronous communication (Slack Inf).

Figure 14 compares the classification accuracy as a function of the number of training images processed, for GeePS with BSP and SSP (Slack 1). The result shows that, when using Slack 1, many more images must be processed to reach the same accuracy as with BSP (e.g., 4.6× more to reach 7%). Since the training throughput with Slack 1 is only 40% higher than with BSP, it is not sufficient to overcome the reduced training quality per image processed—using BSP leads to much faster convergence. We believe that the much higher rate of model parameter updates when using GPUs causes more use of stale values than has been observed for data-parallel CPU-based model training, explaining the difference in results. Interestingly, this outcome 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 converge 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. This section discusses some recent systems using multiple GPUs for deep learning.

Coates et al. [9] describes 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 much 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 [28] is a self-described custom-built supercomputer for deep learning via GPUs. The GPUs used

have large memory capacity (12 GB), and their application fit, allowing use of data-parallel execution. They also support for model-parallel execution, with ideas borrowed from Krizhevsky et al. [19]. They partition the model on fully connected layers, but not on convolutional 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 uses a general parameter server architecture like GeePS.

MXNet [3] (previously CXXNet) is a multi-GPU deep learning system built on the parameter server described by Li et al. [21]. While we are not aware of a detailed description or performance evaluation of that system, we would expect it to perform like the CPU-PS results in Section 5.

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 much bigger than GPU memory, swapping data to and from CPU memory in the background. Combined, GeePS demonstrates that deep learning on GPUs can use data-parallel execution and the general-purpose parameter server model to achieve high-performance deep learning.

References

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

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

[3] MXNet `https://github.com/dmlc/mxnet`.

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

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

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

[7] T. Chilimbi, Y. Suzue, J. Apacible, and K. Kalyanaraman. Project adam: Building an efficient and scalable deep learning training system. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 571–582, 2014.

[8] D. Ciresan, U. Meier, and J. Schmidhuber. Multi-column deep neural networks for image classification. In *Computer Vision and Pattern Recognition (CVPR), 2012 IEEE Conference on*, pages 3642–3649. IEEE, 2012.

[9] A. Coates, B. Huval, T. Wang, D. Wu, B. Catanzaro, and N. Andrew. Deep learning with cots hpc systems. In *Proceedings of the 30th international conference on machine learning*, pages 1337–1345, 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*, pages 37–48, 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, et al. Exploiting iterative-ness for parallel ML computations. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 1–14. ACM, 2014.

[12] G. E. Dahl, D. Yu, L. Deng, and A. Acero. Context-dependent pre-trained deep neural networks for large-vocabulary speech recognition. *Audio, Speech, and Language Processing, IEEE Transactions on*, 20(1):30–42, 2012.

[13] 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 *Advances in Neural Information Processing Systems*, pages 1223–1231, 2012.

[14] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. Imagenet: A large-scale hierarchical image database. In *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*, pages 248–255. IEEE, 2009.

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

[16] 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. *Signal Processing Magazine, IEEE*, 29(6):82–97, 2012.

[17] 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.

[18] 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.

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

[20] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[21] 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*, pages 583–598, 2014.

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

[23] 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 (IJCV)*, pages 1–42, April 2015.

[24] 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.

[25] 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.

[26] 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.

[27] 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 *Proceedings of the Sixth ACM Symposium on Cloud Computing*, pages 381–394. ACM, 2015.

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

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