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

16M58336 Zhou Zixuan
Dept. Civil and Environmental Engineering

Content

Introduction

Large scale deep learning

Large-scale deep learning requires huge computational resources to train a multi-layer neural network.

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

Training on distributed GPUs can be inefficient (data movement overheads, GPU stalls, and limited GPU memory)

GeePS

Introduction

Best Algorithm and Massive Data

More data can improve prediction performance, which also means a more heavy computing burden, computing power will be the biggest bottleneck for AI development in the future.

Depth learning requires massive data, which requires large-scale network model fitting. If the training data is insufficient, it will cause less fitting; and vice verse, only the low precision model will be obtained.

Given sufficient training data and computing power, deep learning approaches far outperform than other approaches for such tasks.

Introduction

Limitation

GPU has a large number of computing units and long lines, and have powerful parallel computing ability and floating-point computation ability, can greatly accelerate the deep learning model training speed, power consumption compared to CPU can provide faster processing speed, less investment and lower server. This also means that the training depth learning model on GPU clusters has shorter iteration times and more frequent parameter synchronization.

The challenges of limited GPU memory and inter-machine communication have been identified as major limitation.

GeePS: a parameter server system

Introduction

Previous Research Comparison

Similarities

GeePS handles the synchronization and communication complexities associated with sharing the model parameters being learned across parallel workers

Differences

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.

Introduction

Data-parallel model training

GeePS supports data-parallel model training

Advantage

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.

Shortcoming

Data-parallel approaches are limited by the desire to fit the entire model in each worker’s memory and this would seem to imply that GPU-based systems (with their limited GPU memory) are suited only for relatively small neural networks.

Introduction

Solution

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.

Single

Experience A state-of-the-art open-source system for deep learning on a single GPU, storing its data in GeePS by 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, as compared to 65% when using an efficient CPU-based parameter server implementation.

Multiple

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

Introduction

Contributions

Describes the first GPU-specialized parameter server design and the changes needed to achieve efficient data-parallel multi-machine deep learning with GPUs.

It reports on large-scale experiments showing that GeePS indeed supports scalable data parallel execution via a parameter server, in contrast to previous expectations.

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.

High performance deep learning

(a) Background of deep learning

(b) GPU architecture

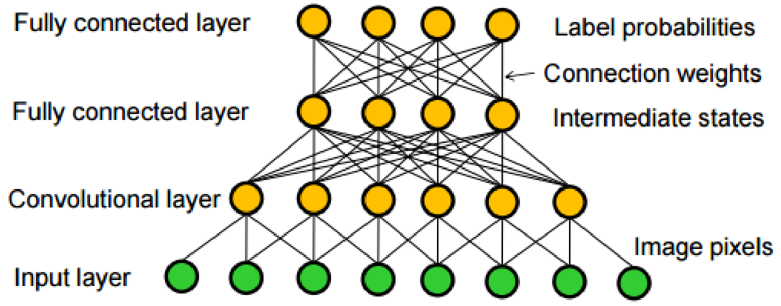
(c) Parameter service of Machine Learning

High performance deep learning

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, which consists of a layered network of nodes and edges (connections), as depicted in Figure.



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

High performance deep learning

Back propagation

1986

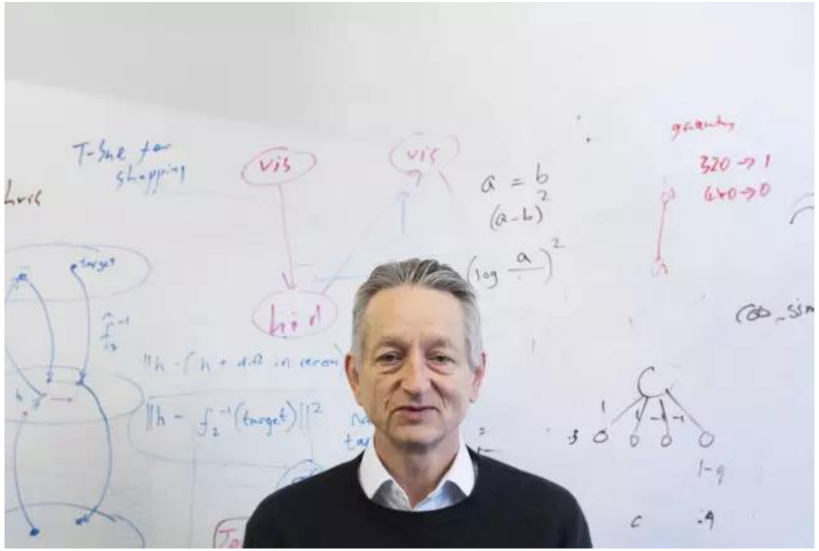
Experiments on learning by back propagation

David C. Plaut, Steven J. Nowlan, Geoffrey E. Hinton

2017

Dynamic Routing Between Capsules

Sara Sabour, Nicholas Frosst, Geoffrey Hinton



High performance deep learning

Stochastic gradient descent (SGD) algorithm

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

Cost function

Optimization objective

Use gradient descent to minimize the cost function

When we have a very large training set, gradient descent becomes a computationally very expensive procedure.

High performance deep learning

Modification to the basic gradient descent algorithm

(Linear regression)

BGD

$$h_\theta(x) = \sum_{j=0}^{n} \theta_j x_j$$

$$J_{train}(\theta) = \frac{1}{2m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2$$

Repeat{

$$\theta_j := \theta_j - \alpha \frac{1}{m}\sum_{i=1}^{m}(h_\theta\left(x^{(i)}\right) - y^{(i)})^2$$

SGD

$$cost(\theta) = \frac{1}{2}(h_\theta(x^{(i)}) - y^{(i)})^2$$

$$J_{train}(\theta) = \frac{1}{m}\sum_{i=1}^{m} cost(\theta, (x^{(i)}, y^{(i)}))$$

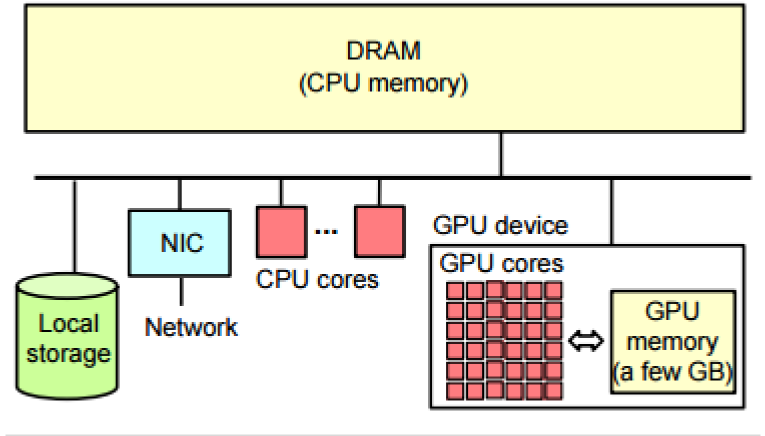
J_{train} is just the average over m training examples of the cost of hypothesis on that example $x^{(i)}, y^{(i)}$

Do not need to look at all the training examples in every single iteration, but needs to look at only a single training example in one iteration.

High performance deep learning

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 GPUs are on self-contained devices that can be inserted into a server machine.



A machine with a GPU device.

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 for it to be accessed efficiently.

High performance deep learning

Limitation in deep learning

Assuming that every 0.5 seconds an iteration, each worker needs to be transmitted over the network by more than 4GB, even if the use of 10GB, parameter synchronization will instantly fill the network. Taking into account that training data may be loaded through NFS or HDFS, it also takes up a lot of network bandwidth. In a data analysis task and AI/ML task mixing environment, large data analysis tasks also consume a lot of network bandwidth (such as shuffle operations), the network delay will be more serious. So if you want to boost your computing power in the Scale out way, the network will be the biggest bottleneck. Previous experiments have proved that Tensorflow distributed training in 8 nodes, and for VGG19 network, 90% of the time spent waiting for network transmission.

Zhang, H., Zheng, Z., Xu, S., Dai, W., Ho, Q., Liang, X., ... & Xing, E. P. (2017). Poseidon: An Efficient Communication Architecture for Distributed Deep Learning on GPU Clusters. *arXiv preprint arXiv:1706.03292*.

High performance deep learning

Method for eliminating network bottleneck

Two modes

Distributed depth learning can take two modes: BSP (Bulk Synchronous Parallel) and SSP (Stale Synchronous Parallel). SSP achieves the effect of balancing computation and network communication overhead by allowing faster worker to use staled parameters.

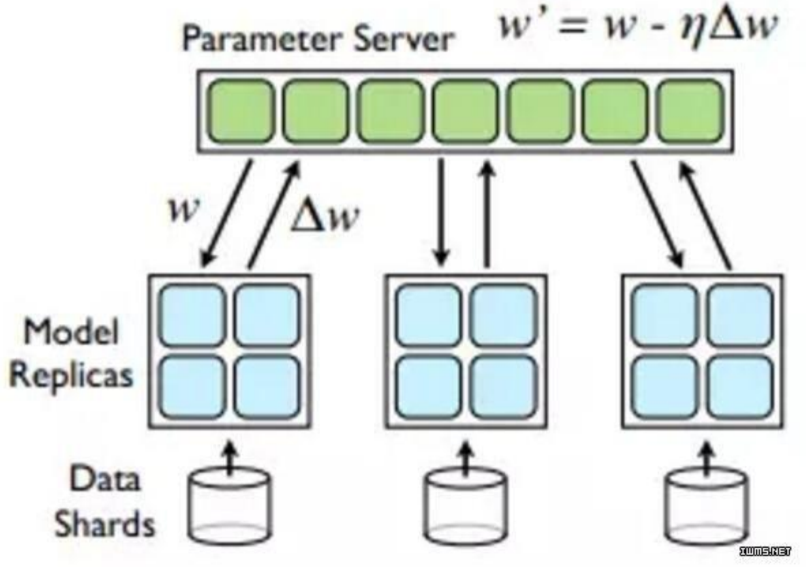
SSP converges slowly at each iteration, but each iteration time is shorter. In the CPU cluster, the overall convergence rate of SSP is faster than that of BSP, but in the GPU cluster training, the overall convergence rate of BSP is much faster than that of SSP.

High performance deep learning

Parameter server

One drawback of the BSP model is that worker needs to send a gradient update to the Parameter Server after each iteration, starting at each iteration, and the worker needs to receive the updated parameters from the Parameter Server, which results in a great deal of network transmission at once.

The parameter server cuts the parameters into block, and shard to several machines, compares all reduce, and effectively uses the network bandwidth to reduce the network delay. At present, the mainstream deep learning systems (Tensorflow, Mxnet, Petuum) all choose to use parameter server to do parameter synchronization.

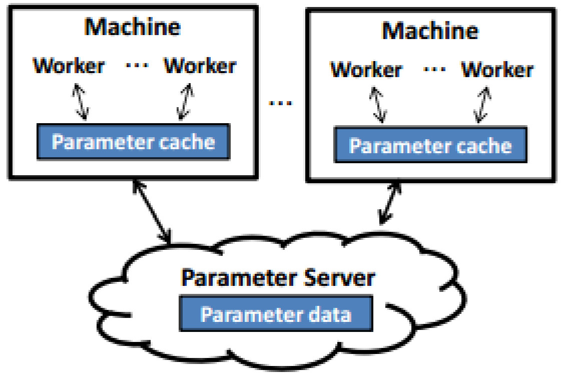


High performance deep learning

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.



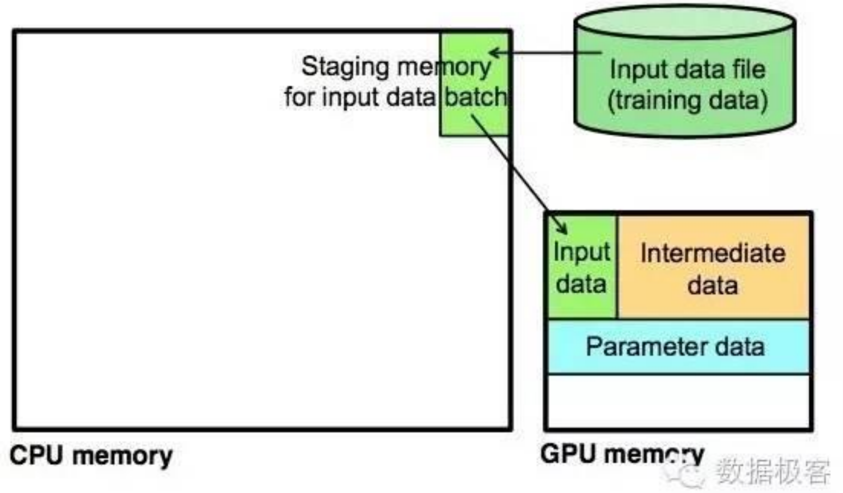
Parallel ML with parameter server.

High performance deep learning

GPU’s memory

One of the features of the GPU device is that it has local memory, and any data needs to be first loaded into the GPU local memory before it can be calculated.

The usual process is: CPU reads a mini-batch training data from the file, moves to GPU memory, then calls cuBLAS and cuDNN libraries from a single worker, and other libraries of the NVIDIA to perform GPU operations.



High performance deep learning

Data transmission

The parameter server is an important architectural model for distributed machine learning.

In the parameter server, the model parameters are stored in a shared distributed Key-Value, which is the parameter server itself.

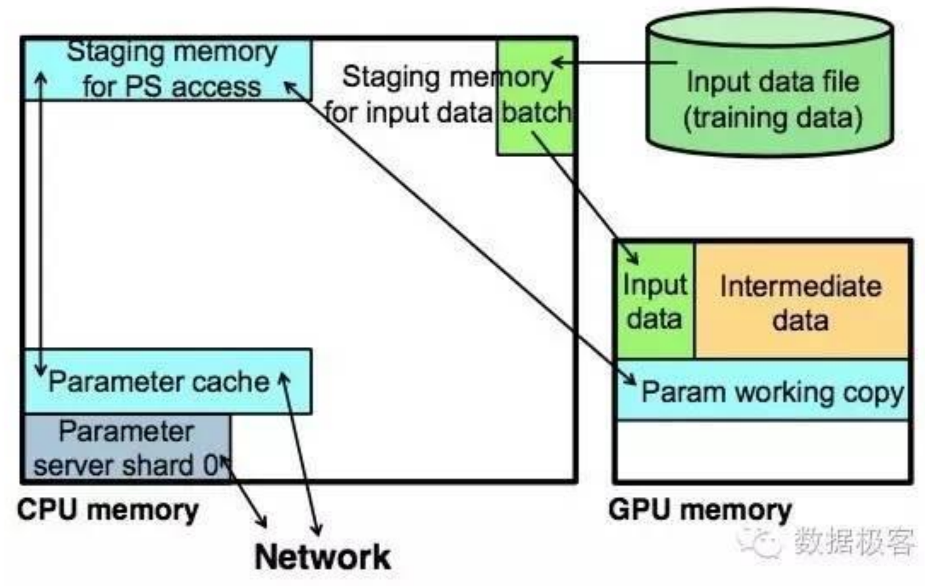
The Worker node that runs the machine learning program uses two interfaces, Read and Update, to communicate with the parameter server, and controls the communication overhead and consistency convergence by the parameter server itself.

GPU-specialized parameter server

Distributed ML on GPUs using a CPU-based parameter server

The design parameters of server are all around the architecture of CPU centered design, as shown in the preceding figure GPU operation limit, if not this part of the factors to take into account the design parameters of the server itself, will cause a higher communication overhead.

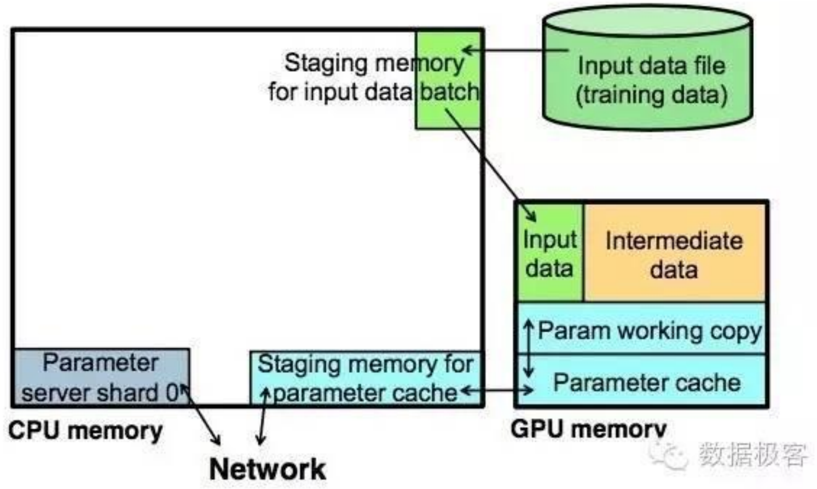
For example, the following figure shows the parameter sharding of the parameter server, stored in CPU memory, and the model parameters running inside the GPU, and also has a local copy in CPU. Therefore, data needs to be moved between the three directions.



GPU-specialized parameter server

Modification

GeePS is a parameter server architecture specifically designed for GPU access. In GeePS, parameters directly stored in GPU, so the architecture of three party data reduction for two-way two-way replication, as shown below; at the same time, GeePS data to move between GPU and CPU is on the background of the implementation.



GPU-specialized parameter server

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.

Movement the parameters storage location of the Key-Value from CPU to GPU can not completely solve the problem, this is because the GPU data will be very slow, so in general the parameters of server architecture, which will greatly reduce the throughput due to synchronous read each Key-Value.

GPU-specialized parameter server

Parallelizing batched access

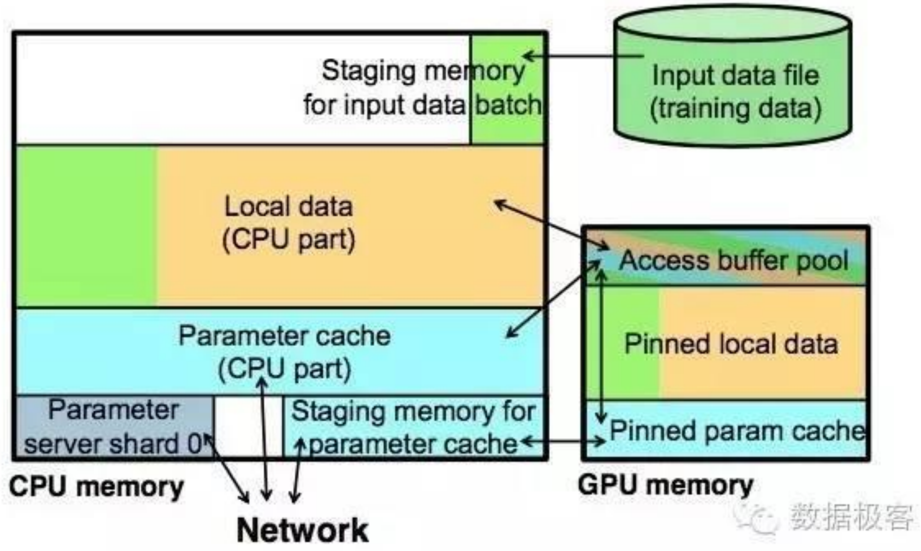
Further work done by GeePS is the introduction of batch operation parameters, GPU memory management, the parameters of the model do not need currently migrate to CPU through a background thread, thereby minimizing synchronization overhead caused by waiting for data transmission.

In this way, 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.

GPU-specialized parameter server

Parallelizing batched access

Specifically, compared to the general parameters of Read server and Update interface, providing additional GeePS's PostRead and PreUpdate two interface, when the application needs to read parameters, parameter server allocates a block buffer in GPU memory, and returns the pointer, read after the end, responsible for the release of buffer by the PostRead, the interface is non blocking operation.

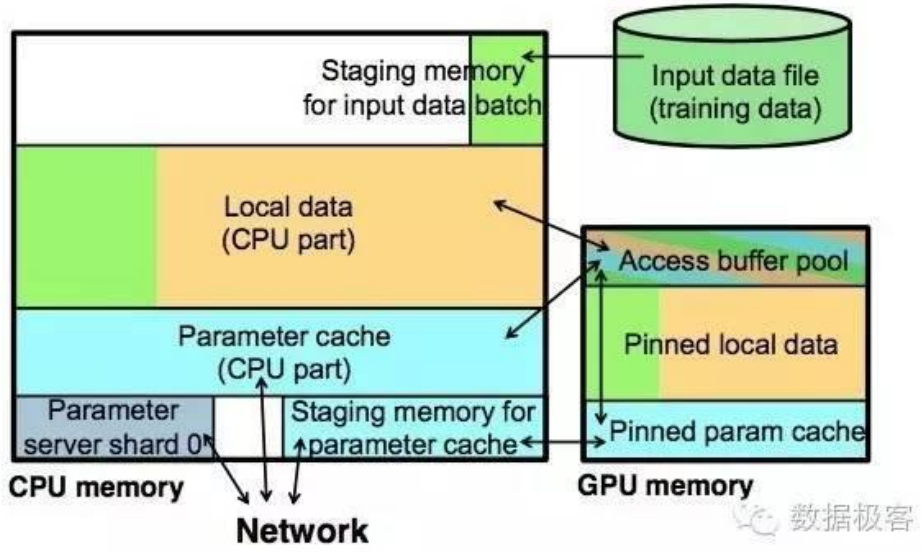


GPU-specialized parameter server

Parallelizing batched access

When the application needs to update the parameters, first obtained by pre-update buffer, and non blocking update interface is responsible for updating the parameters and release buffer.

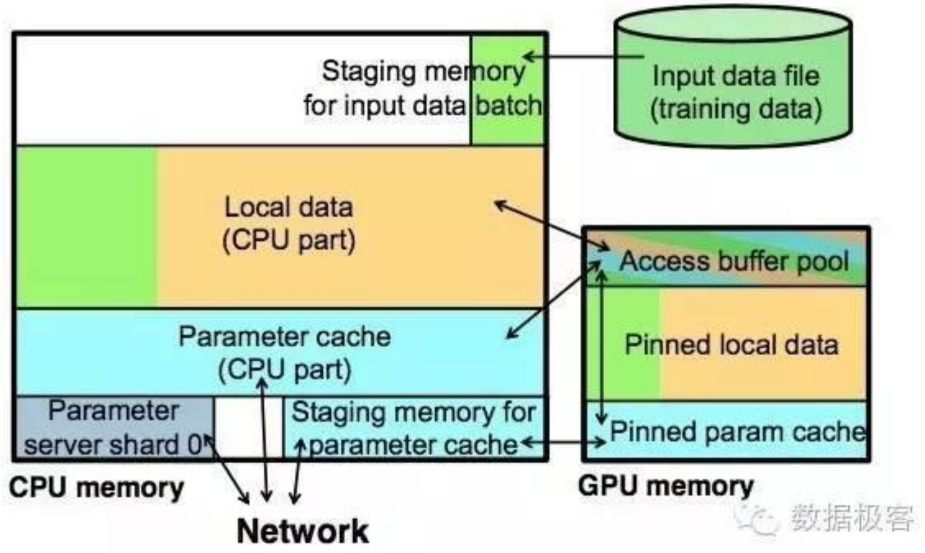
Therefore, the idea is simple, using non blocking I/O to exchange data between different memory, improve throughput, and avoid the performance problems caused by the synchronous overhead of single record reading.



GPU-specialized parameter server

Parallelizing batched access

The design of the buffer GeePS, actually management from the entire GPU memory, because GPU memory size is limited, the parameters of Key-Value server in the GPU memory is clearly not realistic, do not stop by in the background between CPU and GPU for data exchange, so as to create the probability of storage huge model parameters in a small amount of GPUs the memory.



Design of data exchange based on buffer

GPU-specialized parameter server

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.

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).

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.

GPU-specialized parameter server

Data placement policy

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.

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.

GPU-specialized parameter server

Data placement policy

(a) Pinning local data in GPU memory gives move benefit than pinning parameter cache data.

(b) If the data is used at the peak usage, the usage of peak access buffer can be reduced.

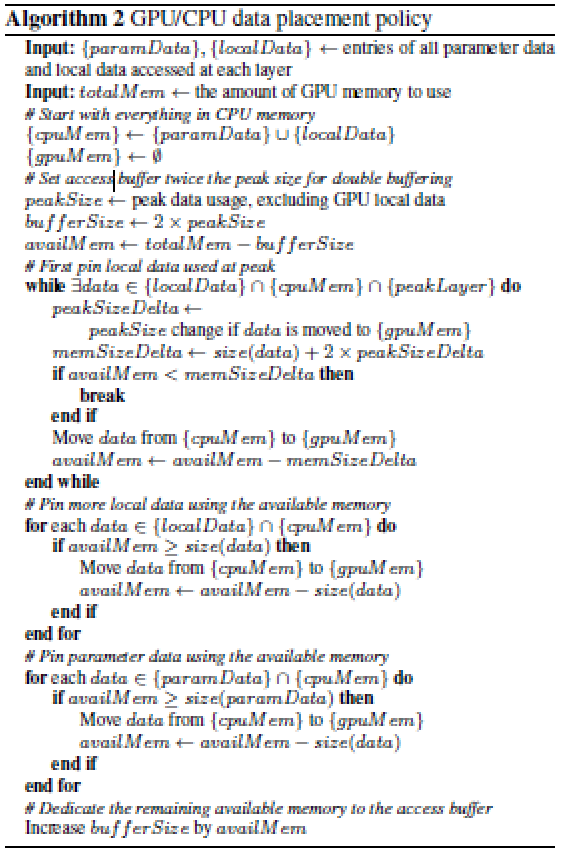
GPU-specialized parameter server

Algorithm

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.

Second, it will try to use the available capacity to pin more local data and parameter cache data in GPU memory.

Third, it will add any remaining available GPU memory to 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
$peakSize$ ← 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
    $peakSizeDelta$ ←
        $peakSize$ 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 $availMem$
```

GPU-specialized parameter server

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.

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.

Evaluation

Summary

The idea and implementation of GeePS is not complicated. It is the introduction of distributed parameter caching using the characteristics of GPU data access.

According to the author, GeePS in the small and medium sized GPU clusters made to enhance the performance of near linear 16 node cluster relative to a single machine up to 13 times, compared with the design parameters of the server based on CPU was about 2 times the throughput.

Evaluation

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 large 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.

Evaluation

Scaling deep learning with GeePS

Results of comparison in data-parallel scaling training on both image classification and video classification.

(1) GeePS and Single-GPU

(2) GeePS and GPU workers with CPU-based parameter server.

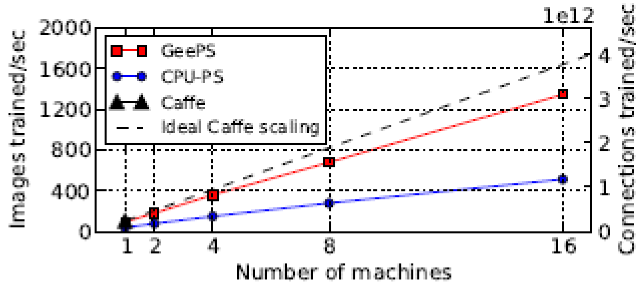
(3) CPU workers with CPU-based parameter server

Evaluation

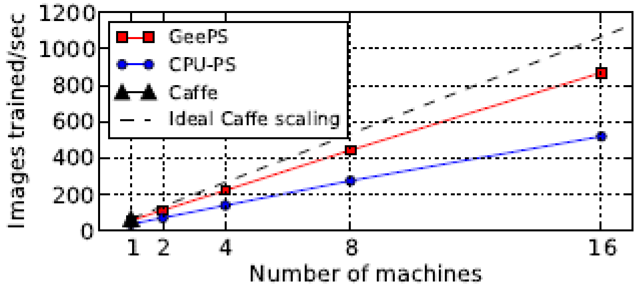
Scaling deep learning with GeePS

Image classification

GeePS scales almost linearly when we add more machines. Compared to the single-machine optimized Caffe, GeePS achieves 13 × speedups on both GoogLeNet and AdamLike model with 16 machines. Compared to CPU-PS, GeePS achieves over 2 × more throughput. The GPU stall time of GeePS is 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.



(a) AdamLike model on ImageNet22K dataset.



(b) GoogLeNet model on ILSVRC12 dataset.

Evaluation

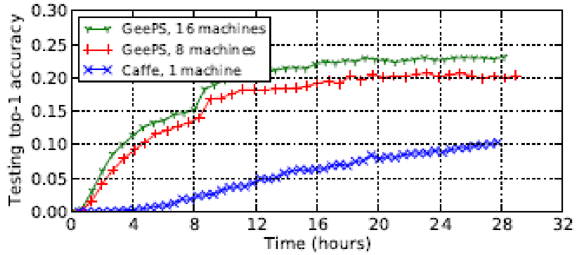
Scaling deep learning with GeePS

Accuracy

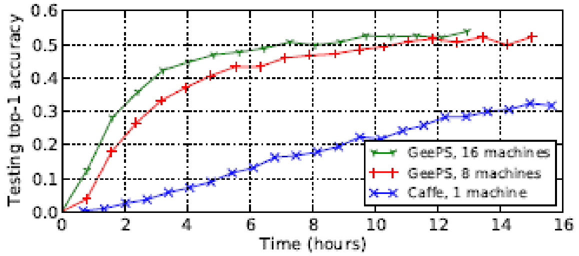
Definition: The fraction of the testing images that are correctly classified.

Comparing the amount of time required to reach a given level of accuracy so that evaluate convergence speed, which is a combination of image training throughput and model convergence per trained image.

Caffe needs 13.7 hours to reach 30% accuracy, while GeePS needs only 2.8 hours with 8 machines or 1.8 hours with 16 machines.



(a) AdamLike model on ImageNet22K dataset.



(b) GoogLeNet model on ILSVRC12 dataset.

Image classification top-1 accuracies.

Evaluation

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.

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.

GeePS enables use of data-parallel execution and the general-purpose parameter server model to achieve efficient, scalable deep learning on distributed GPUs.

Thank you