Need for Co-Design

CSE 5194.01
Autumn ‘19

Bharath Ramesh
The Ohio State University
E-mail: ramesh.113@osu.edu
Outline

• Introduction
• S-Caffe
• Horovod
• Conclusion
What is Co-Design?

• Designing communication runtimes taking DL frameworks into account to provide optimal “scale-out” performance.

• Two ways of doing Co-Design
  – **Tightly Coupled**: Rigorous Co-design of both communication run-time and DL frameworks. Requires modification of the framework as well as the communication runtime. Example: S-Caffe
  – **Loosely Coupled**: Treat DL framework as a “black-box”. Creates wrappers around other DL frameworks and utilizes communication operations such as Allreduce for efficient parallelization. Example: Horovod.
The Need for Co-Design

- Larger and Deeper models are being proposed that are computationally expensive
- Single GPU training is relatively slow
- Multi-GPU in one node is good but there is a limit to Scale-up (as mentioned before)
- **Multi-node (Distributed or Parallel) Training** requires optimizing communication runtimes
- Emergence of DL frameworks such as tensorflow, caffe
- In some cases, optimizing a framework’s workflow for HPC is beneficial
Outline

• Introduction

• S-Caffe

• Horovod

• Conclusion
S-Caffe: Co-designing MPI Runtimes and Caffe for Scalable Deep Learning on Modern GPU Clusters

A. A. Awan, K. Hamidouche, J. M. Hashmi, and D. K. Panda
In Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP ‘17)

https://dl.acm.org/citation.cfm?id=3018769

Note: Images and some text used in the S-Caffe section are taken from the authors of the paper
CUDA-Aware MPI: Before and After

- Before CUDA 4.0, lack of a common memory registration mechanism
  - Each device has to pin the host memory
  - Many operating systems do not allow multiple devices to register the same memory pages
  - Previous solution: Use different buffer for each device and copy the data

- After CUDA 4.0, both devices register a common host buffer
  - GPU copies data to this buffer, and the network adapter can directly read from this buffer (or vice-versa)
  - Note that GPU-Direct does not allow you to bypass host memory
CUDA-Aware MPI : Before and After

- After CUDA 5.0 (GDR), network adapter can directly read/write data from/to GPU device memory
- Avoids copies through the host all together
- Fastest possible comm. between GPU and IB HCA
- Allows for better asynchronous communication
MPI_Communicators

- Form a logical group of processes that have a context associated with them
- Collective operations can be called on a per communicator basis
- Very useful to build “topology” aware hierarchical collectives
- MPI_Comm_Split is called with a “color” which serves as an identifier for the group
- Applications can have many communicators that exploit architectures taking communication patterns into account

Source: https://mpitutorial.com/tutorials/introduction-to-groups-and-communicators/
Research Challenges

- What are the performance and portability bottlenecks in the existing DL frameworks?
  - Why do we need to support distributed training?

- How to design a Scalable and Distributed Framework?
  - Achieve both Scale-up and Scale-out efficiently?

- What are the new requirements and expectations for Communication Runtimes?
  - What are specific challenges for the MPI runtimes?

- Can a Co-design approach help in achieving better training performance?
  - Can a DL framework like Caffe be co-designed with an MPI runtime?

- What is the impact of the co-design?
  - What performance benefits can be observed? At what levels?
Research Challenges - Bottlenecks for MPI runtimes

- Bringing in “GPU” awareness to efficiently handle buffers that are computed by the GPU instead of “staging” through the host
- Parameter server performs a large broadcast to distribute model parameters
- Large reductions in the gradient aggregation step
- Very large buffers (often > 4MB) saturate the bandwidth and might lead to contention in the network

http://arxiv.org/abs/1511.00175
Research Challenges – Co-Design

- cuDNN, cuBLAS and NCCL give great performance for 1 node scenarios (scale-up performance, up to 16 GPUs in DGX-2)
- CUDA-aware MPI and NCCL2 communication runtimes provide optimal scale-out performance
- Goal of Co-Design – To achieve both by:
  - Overlap of computation and communication
  - Implementing efficient Large-message collectives (specifically reduce)
  - Optimizing application workflows to exploit communication runtime co-designs
Caffe Architecture

Phase 1:
Data propagation uses broadcast for DNN parameter exchange

Phase 2:
Forward/Backward pass involves computation

Phase 3:
Gradient aggregation involves reduce to aggregate gradients
Caffe Architecture

- Phases are blocking in nature
  - Some are blocking due to true data dependencies
  - Most work in a non-blocking (overlapping) manner
- **Idea of S-Caffe**: Adopt a fine-grained workflow to overlap phases

```
• Phases are blocking in nature
  • Some are blocking due to true data dependencies
  • Most work in a non-blocking (overlapping) manner
• Idea of S-Caffe: Adopt a fine-grained workflow to overlap phases
```
S-Caffe idea overview

- Co-Design MPI (communication) and application (DL framework) runtimes to provide optimal performance

- At the MPI level:
  - Optimize support for large message reductions of GPU buffers
  - Provide non-blocking operations to ensure maximal overlap of computation and communication

- At the application level:
  - Change workflow to follow a fine-grain approach, which does “layer-wise” communication instead of communicating the entire set of model parameters

- Provides 3 designs SC-Basic (SC-B), SC-Optimized broadcast (SC-OB) and SC-Optimized broadcast and reduce (SC-OBR)
S-Caffe architecture
Design 1 – S-Caffe basic (SC–B)

• Optimize explicit data movements
  – Move from “shared” data access to distributed data access
  – Communicate GPU based buffers directly in a “CUDA-aware” fashion
  – Avoids unnecessary copies to host buffers

• Collective operations for data propagation and aggregation
  – Replace naïve point to point sends in data propagation with CUDA-aware MPI_Bcast
  – Replace point to point operations in data aggregation with CUDA-aware MPI_Reduce
  – The above could work very well for small scale jobs and relatively small datasets, but is limited in scalability
Design 1 – S-Caffe basic (SC–B)

- Training data needs to be loaded from disk, which is relatively slow
- Could use MPI to exchange data from one node to other nodes but is slower on parallel read capable filesystems
- Use parallel data reader threads
  - For each of the processes and
  - Maintain separate distributed queues
  - Advantageous in parallel filesystems such as lustre
Design 2 – S-Caffe optimized broadcast (SC–OB)

• SC-B becomes “communication bound” for large datasets/ large number of models
• Caffe performs communication and computation in sequential phases
  – Utilize NBC (non-blocking collective operations) from MPI-3 to overlap computation and communication
  – Introduce fine-grained phases for each layer to achieve overlap
• Replace MPI_Bcast() calls in data propagation with MPI_Ibcast()
• Efficient overlap requires careful use of the MPI_Wait() operation
  – Calling this too soon will lead to poor computation/communication overlap as the communication will be mainly progressed during this call.
  – Calling it too late might lead to degradation as it limits the potential of overlapping future computation phases
Design 2 – S-Caffe optimized broadcast (SC–OB)

- Naïve NBC data propagation design
  - Overlap communication of layer “i+1” with computation of layer i
  - Calls Ibcast and Wait before the layer that requires the data
  - Limits asynchronous progress
- Can we do better?
Design 2 – S-Caffe optimized broadcast (SC–OB)

- Start all Ibcast operations at the beginning
- Call the Wait operation of ith Ibcast just before the ith Forward pass of a layer that actually needs the corresponding data
- Provides maximal overlap in the data propagation step
Design 3 – S-Caffe optimized gradient aggregation (SC–OBR)

• Same method used in SC-OB won’t work for gradient aggregation due to
  – MPI not having effective Ireduce operations as the CPU is required to do computation and communication thereby limiting overlap
  – falling back to a layer-wise blocking reduce operation will not provide any overlap
    • the number of steps in performing N medium sized reductions are equivalent to a single big reduction
    • Backpropagation is sequential from layer to layer
• Requires careful co-design to overlap gradient aggregation with backward pass
• Proposed designs include a “helper thread” and a “DL-aware” hierarchical reduce
Design 3 – S-Caffe optimized gradient aggregation (SC–OBR)

- Why does MPI not have an effective Ireduce?
  - MPI runtimes are unaware of user’s CUDA streams
  - No way to know whether GPU buffer pointer passed to MPI has a pending computation on a stream writing to the buffer
  - Requires explicit synchronization of stream to ensure correctness
  - No way for streams to wait for blocking operations (such as MPI_Wait)
  - Other streams that synchronize with the first stream might be blocked due to the above
Design 3 – S-Caffe optimized gradient aggregation (SC–OBR)

• Offload invocation of backward pass to helper thread
• Helper thread signals main thread to invoke reductions
  – Uses C++ condition flags
• Layered design doing reduce for Layer N after backward pass for layer N
• Provides overlap between computation (backward pass) of Layer N-1 with Reduce of Layer N
Hierarchical reduce (HR)

- Intra and inter-node hierarchy not enough in GPU’s context (one node has 2-16 GPUs only)
- Hierarchy should include two levels
  - A binomial tree and lower level communicator spanning a few nodes
  - Algorithms in both levels can be tuned at runtime
- Number of GPUs in the lower level communicator = chain length
- Algorithm divides buffer into N chunks and sends each chunk to the left. Leads to overlap of reduction and communication for chunks
- Optimal chain length found = 8 for message sizes >=8MB
Hierarchical reduce mathematical model

- Let, $P =$ number of processes ,
- $n =$ number of chunks
- $t(b) =$ time to reduce buffer of size $b$
- $t(c) =$ time to reduce buffer of size $c \ [c = b/n]$
- $T(CC) =$ Total Time for Chunked Chain Algorithm
- $T(Bin) =$ Total Time for Binomial Tree Algorithm

- $T(Bin) = \log(P) \ast t(b) \quad (1)$
- $T(CC) = (P - 1) \ast t(c) + (n - 1) \ast t(c)$
- $T(CC) = (n + P - 2) \ast t(c) \quad (2)$

Analyzing equations (1) and (2) for different number of processes $P$ and buffer size $b$, we observe:
- for small $P$ and large $b$, $T(CC) \ll T(Bin)$
- But, for large $P$ and small $b$ $T(CC) \gg T(Bin)$
Results for Hierarchical Reduce (160 GPUs)
Results for Hierarchical Reduce (160 GPUs)

- MVAPICH2
- Proposed
- OpenMPI

Latency (sec)

Message Size (MB)

- > 100x
- 2.5x
Results for S-Caffe: CIFAR 10 (64 GPUs)

- CIFAR 10 is a medium sized model
- CIFAR solve run for 1,000 iterations with batch-size 8,192
- Increased batch-size can demonstrate scale-out
- Caffe scales only up to 16 GPUs (max on one node)
- “Strong scaling” numbers are shown (where the problem size is fixed but the solvers are increased. Example: if batch size = 1024 and Num GPUs = 32, then effective batch size is 32/ GPU)
Comparison (AlexNet): S-Caffe, Inspur-Caffe, and CNTK

- Alexnet contains 64 million parameters and buffers of 256MB
- Inspur Caffe – Parameter server based MPI
- CNTK – Microsoft’s toolkit that uses MPI
- Higher samples per second indicates better performance
- Communication overhead is significant
- S-Caffe provides state-of-the-art performance
S-Caffe: GoogLeNet (160 GPUs)

- GoogleNet contains 12 million parameters and buffers > 50MB
- Communication overhead is significant
- S-Caffe provides scale-out to 160 GPUs
- S-Caffe-L uses LMDB database whereas S-Caffe uses the Lustre file system.
S-Caffe Conclusion

- S-Caffe: rigorous co-designs at the application level as well as the runtime level
  - Provides a scale-out up to 160 GPUs
  - Exploits CUDA-Aware communication including reductions
  - Proposes “fine-grained” communication approach to ensure overlap
  - Presents methodology that can be extended to other DL frameworks
  - Achieves speed-ups better than all other distributed frameworks
- All designs are publicly available at http://hidl.cse.ohio-state.edu
Outline

- Introduction
- S-Caffe
- Horovod
- Conclusion
Horovod: fast and easy distributed deep learning in TensorFlow

Alexander Sergeev, Mike Del Balso

https://arxiv.org/abs/1802.05799

Note: Source of images in this section: https://eng.uber.com/horovod/
Introduction to Horovod

• Designed by Uber as a part of their “Michelangelo” platform (an ML-as-a-service platform to build and deploy machine learning systems).

• Exists to make it easier to create and run distributed deep learning projects
  – Initially started with TensorFlow and keras
  – Later extended to other frameworks as well

• Named after the Russian folk dance “horovod” where dancers perform in a circle and thereby resembling how distributed TensorFlow processes use Horovod to communicate with each other
Introduction to Horovod

• The great results obtained at the time by using a “data-parallel” approach motivated the creation of horovod
• Gradients from each node are averaged from every iteration
• Very useful for models which fit into one GPU
Why not just use Distributed Tensorflow?

- Not clear on the correct set of code modifications required to distribute model training code
- Coding using workers, parameter servers, tf.ClusterSpec() and others introduced subtle and hard to diagnose bugs
- Could not scale very well for large clusters
Why not just use Distributed Tensorflow?

- Communication overhead significant at 128 GPUs
- Distributed tensorflow couldn’t utilize half the GPU resources at that scale
Why not just use Distributed Tensorflow?

- Difficult to identify the right ratio of workers to parameter servers
- Increased code complexity due to the parameter server model
  - Required user to perform explicit passing of service discovery information such as hosts and ports of all the workers and parameter servers and modify the training program to construct tf.Server() with an appropriate tf.ClusterSpec().
  - Additionally, users had to ensure that all the operations were placed appropriately using replica setters
  - Steep learning curve
Solution: Horovod!

- Convert code to standalone python package for ease of install
- Use “ring Allreduce” at it’s core to average gradients in the form of communication libraries such as NCCL2 and MPI
- Added support for models that fit inside a single server, potentially on multiple GPUs, whereas the original version only supported models that fit on a single GPU
- Implemented broadcast for consistent initialization of model on worker nodes
Horovod Stack

- Plugs into tensorflow directly via “custom ops”
- Can use MPI/NCCL/ other communication libraries for reductions and worker coordinations
Horovod Code Example

- Hvd.init() initializes the framework
- Visible_device list assigns GPU to tensorflow process
- hvd.DistributedOptimizer(opt) replaces the tf optimizer with an “all-reduce” call to all gradients
- hvd.BroadcastGlobalVariablesHook(0) broadcasts variables from the first process to all other processes to ensure consistent initialization

```python
import tensorflow as tf
import horovod.tensorflow as hvd

# Initialize Horovod
hvd.init()

# Pin GPU to be used to process local rank (one GPU per process)
config = tf.ConfigProto()
config.gpu_options.visible_device_list = str(hvd.local_rank())

# Build model...
loss = ...
opt = tf.train.AdagradOptimizer(0.01)

# Add Horovod Distributed Optimizer
opt = hvd.DistributedOptimizer(opt)

# Add hook to broadcast variables from rank 0 to all other processes during # initialization.
hooks = [hvd.BroadcastGlobalVariablesHook(0)]

# Make training operation
train_op = opt.minimize(loss)

# The MonitoredTrainingSession takes care of session initialization, # restoring from a checkpoint, saving to a checkpoint, and closing when done # or an error occurs.
with tf.train.MonitoredTrainingSession(checkpoint_dir="/tmp/train_logs",
    config=config,
    hooks=hooks) as mon_sess:
    while not mon_sess.should_stop():
        # Perform synchronous training.
        mon_sess.run(train_op)
```
Horovod Code Example

  - Trains model across 16 GPUs, with 4 GPUs per node
  - MPI takes care of communication whereas horovod manages everything else with tensorflow as a back-end
  - Can also use `horovodrun -np 4 -H localhost:4 python train.py`
    - Horovodrun provides a convenient wrapper around MPI calls
- Reduced the amount of boilerplate code need to distribute job. In this case, only 4 extra lines were needed!
Ring-Based Allreduce
Ring-Based Allreduce

• Consider an all-reduce operation on X items with the size of each item being itsize \( (m\text{size} = X \times \text{itsize}) \)
• Communication time is \( 2 \times (N - 1) \times T\left((X \times \text{itsize}) / N\right) \), where \( N \) is the number of processes
• Proven to be bandwidth optimal for large messages!!
• Used for large message sizes (> 2MB or so)

Horovod Timeline

• Used to easily find bugs in code
• Each tensor reduction has 2 major phases
• **Negotiation** - a phase when all workers send to rank 0 signal that they’re ready to reduce the given tensor.
  – Each worker reporting readiness is represented by a tick under the NEGOTIATE_ALLREDUCE bar, so you can see which workers were early and which were late.
  – Immediately after negotiation, rank 0 sends all other workers signal to start reducing the tensor.
Horovod Timeline

- **Processing** - a phase when the operation actually happens. It is further subdivided into multiple sub-phases:
  - **WAIT_FOR_DATA** indicates time taken to wait for GPU to finish computing input to the allreduce, allgather, or broadcast operations. This happens because TensorFlow tries to smartly interleave scheduling and GPU computation. This is only applicable to situations where the Horovod operation is placed on GPU.
  - **WAIT_FOR_OTHER_TENSOR_DATA** indicates time taken to wait for GPU to finish computing other inputs for other operations that are part of the same fusion batch.
  - **QUEUE** happens when reduction is done with NCCL, and the previous NCCL operation did not finish yet.
  - **MEMCPY_IN_FUSION_BUFFER** and **MEMCPY_OUT_FUSION_BUFFER** indicate time taken to copy data into and out of the fusion buffer.
  - **NCCL_ALLREDUCE**, **MPI_ALLREDUCE**, **MPI_ALLGATHER**, or **MPI_BCAST** indicate time taken to do the actual operation on GPU (or CPU) and highlights whether the operation was performed using NCCL or pure MPI.
Horovod Timeline
Horovod Timeline

• Tensorflow timelines and CUDA profilers don’t work because user has to collect and reference profiles from various servers
• Horovod timeline gives tracing support and is compatible with chrome’s about:tracing profiling viewer
• Users can use Horovod Timelines to view exactly what each node was doing at each time step throughout a training job
• Use chrome://tracing to view results
Tensor Fusion

- Many models such as ResNet101 contained large amount of tensors and used tiny allreduce operations
- Fusion “fuses” tensors together before calling Horovod allreduce
- Required to utilize the best out of ring allreduce (as it is optimal only for large messages)
- Provides 65% improvement for models running on unoptimized TCP networks
Tensor Fusion

- The algorithm is as follows:
  - Determine which tensors are ready to be reduced. Select the first few tensors that fit in the buffer and have the same data type.
  - Allocate a fusion buffer if it was not previously allocated. Default fusion buffer size is 64 MB.
  - Copy data of selected tensors into the fusion buffer.
  - Execute the allreduce operation on the fusion buffer.
  - Copy data from the fusion buffer into the output tensors.
  - Repeat until there are no more tensors to reduce in the cycle.
Horovod performs work in cycles. These cycles are used to aid Tensor Fusion. Horovod has the ability to record the moment when each cycle starts for debugging of Tensor Fusion.
Horovod Collectives used (other than allreduce)

- MPI_Allgather – Gathers data from all processes on every process. Used to collect values for sparse tensors

- MPI_Bcast – Broadcast data from one process to all processes. Used excessively in data parallelism (to broadcast model parameters and other details)
Horovod Autotuning

- Horovod contains multiple “knobs” that can affect runtime performance
  - `--cycle-time-ms` and `--fusion-threshold-mb` for tensor fusion
  - `--hierarchical-allreduce` and `--hierarchical-allgather` for collective operations
- Determining the best combination of these values to maximize performance (minimize time to convergence) can be a matter of trial-and-error
- Many factors including model complexity, network bandwidth, GPU memory, etc. can all affect inputs per second throughput during training.
- Solution: autotuning
  - Horovod collects metrics such as bytes of allreduce/unit of time until experiment reaches convergence or after a set number of metrics are collected
  - The result will be used for the rest of the experiment
Horovod Results

- Results shown by running tensorflow benchmarks
- Shows excellent scaling when compared to distributed tensorflow
Horovod Results

- Shows comparison between TCP and RDMA based horovod
- RDMA is significantly better at large scales due to less protocol overheads and high performance interconnects
Outline

• Introduction
• S-Caffe
• Horovod

• Conclusion
Conclusion

• Co-Design helps drastically improve scale-out performance
• Two ways of accomplishing Co-Design
• There are advantages/disadvantages to each method listed above. Tradeoff is between ease of implementation and performance
• Ring based algorithms for communication are effective for large messages and hence many DL applications
• Questions?