High-Performance Models

This document and accompanying scripts (https://github.com/tensorflow/benchmarks/tree/master/scripts/tf_cnn_benchmarks) detail how to build highly scalable models that target a variety of system types and network topologies. The techniques in this document utilize some low-level TensorFlow Python primitives. In the future, many of these techniques will be incorporated into high-level APIs.

Input Pipeline

The Performance Guide (https://www.tensorflow.org/performance/performance_guide) explains how to identify possible input pipeline issues and best practices. We found that using `tf.FIFOQueue` (https://www.tensorflow.org/api_docs/python/tf/FIFOQueue) and `tf.train.queue_runner` (https://www.tensorflow.org/api_docs/python/tf/train/queue_runner) could not saturate multiple current generation GPUs when using large inputs and processing with higher samples per second, such as training ImageNet with AlexNet (http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf). This is due to the use of Python threads as its underlying implementation. The overhead of Python threads is too large.

Another approach, which we have implemented in the scripts (https://github.com/tensorflow/benchmarks/tree/master/scripts/tf_cnn_benchmarks), is to build an input pipeline using the native parallelism in TensorFlow. Our implementation is made up of 3 stages:

- I/O reads: Choose and read image files from disk.
- Image Processing: Decode image records into images, preprocess, and organize into mini-batches.
- CPU-to-GPU Data Transfer: Transfer images from CPU to GPU.

The dominant part of each stage is executed in parallel with the other stages using `data_flow_ops.StagingArea`. `StagingArea` is a queue-like operator similar to `tf.FIFOQueue` (https://www.tensorflow.org/api_docs/python/tf/FIFOQueue). The difference is that `StagingArea` does not guarantee FIFO ordering, but offers simpler functionality and can be executed on both CPU and GPU in parallel with other stages. Breaking the input pipeline into 3 stages that operate independently in parallel is scalable and takes full advantage of large multi-core environments. The rest of this section details the stages followed by details about using `data_flow_ops.StagingArea`.

Parallelize I/O Reads

`data_flow_ops.RecordInput` is used to parallelize reading from disk. Given a list of input files representing TFRecords, `RecordInput` continuously reads records using background threads. The records are placed into its own large internal pool and when it has loaded at least half of its capacity, it produces output tensors.

This op has its own internal threads that are dominated by I/O time that consume minimal CPU, which allows it to run smoothly in parallel with the rest of the model.

Parallelize Image Processing

After images are read from `RecordInput` they are passed as tensors to the image processing pipeline. To make the image processing pipeline easier to explain, assume that the input pipeline is targeting 8 GPUs with a batch size of 256 (32 per GPU).

256 records are read and processed individually in parallel. This starts with 256 independent `RecordInput` read ops in the graph. Each read op is followed by an identical set of ops for image preprocessing that are considered independent and executed in parallel. The image preprocessing ops include operations such as image decoding, distortion, and resizing.

Once the images are through preprocessing, they are concatenated together into 8 tensors each with a batch-size of 32. Rather than using `tf.concat` (https://www.tensorflow.org/api_docs/python/tf/concat) for this purpose, which is implemented as a single op that waits for all the inputs to be ready before concatenating them together, `tf.parallel_stack` (https://www.tensorflow.org/api_docs/python/tf/parallel_stack) is used. `tf.parallel_stack` (https://www.tensorflow.org/api_docs/python/tf/parallel_stack) allocates an uninitialized tensor as an output, and each input tensor is written to its designated portion of the output tensor as soon as the input is available.

When all the input tensors are finished, the output tensor is passed along in the graph. This effectively hides all the memory latency with the long tail of producing all the input tensors.

Parallelize CPU-to-GPU Data Transfer

Continuing with the assumption that the target is 8 GPUs with a batch size of 256 (32 per GPU).

Once the input images are processed and concatenated together by the CPU, we have 8 tensors each with a batch-size of 32.

TensorFlow enables tensors from one device to be used on any other device directly. TensorFlow inserts implicit copies to make the tensors available on any devices where they are used. The runtime schedules the copy between devices to run before the tensors are actually used. However, if the copy cannot finish in time, the computation that needs those tensors will stall and result in decreased performance.

In this implementation, `data_flow_ops.StagingArea` is used to explicitly schedule the copy in parallel. The end result is that when computation starts on the GPU, all the tensors are already available.

Software Pipelining

With all the stages capable of being driven by different processors, `data_flow_ops.StagingArea` is used between them so they run in parallel. `StagingArea` is a queue-like operator similar to `tf.FIFOQueue` (https://www.tensorflow.org/api_docs/python/tf/FIFOQueue) that offers simpler functionalities that can be executed on both CPU and GPU.

Before the model starts running all the stages, the input pipeline stages are warmed up to prime the staging buffers in between with one set of data. During each run step, one set of data is read from the staging buffers at the beginning of each stage, and one set is pushed at the end.

For example: if there are three stages: A, B and C. There are two staging areas in between: S1 and S2. During the warm up, we run:

```
Warm up:
Step 1: A0
Step 2: A1  B0

Actual execution:
Step 3: A2  B1  C0
Step 4: A3  B2  C1
Step 5: A4  B3  C2
```

After the warm up, S1 and S2 each have one set of data in them. For each step of the actual execution, one set of data is consumed from each staging area, and one set is added to each.

Benefits of using this scheme:

- All stages are non-blocking, since the staging areas always have one set of data after the warm up.
- Each stage can run in parallel since they can all start immediately.
- The staging buffers have a fixed memory overhead. They will have at most one extra set of data.
- Only a single`session.run()` call is needed to run all stages of the step, which makes profiling and debugging much easier.

Best Practices in Building High-Performance Models

Collected below are a couple of additional best practices that can improve performance and increase the flexibility of models.

Build the model with both NHWC and NCHW

Most TensorFlow operations used by a CNN support both NHWC and NCHW data format. On GPU, NCHW is faster. But on CPU, NHWC is sometimes faster.

Building a model to support both data formats keeps the model flexible and capable of operating optimally regardless of platform. Most TensorFlow operations used by a CNN support both NHWC and NCHW data formats. The benchmark script was written to support both NCHW and NHWC. NCHW should always be used when training with GPUs. NHWC is sometimes faster on CPU. A flexible model can be trained on GPUs using NCHW with inference done on CPU using NHWC with the weights obtained from training.

Use Fused Batch-Normalization

The default batch-normalization in TensorFlow is implemented as composite operations. This is very general, but often leads to suboptimal performance. An alternative is to use fused batch-normalization which often has much better performance on GPU. Below is an example of using `tf.contrib.layers.batch_norm` (https://www.tensorflow.org/api_docs/python/tf/contrib/layers/batch_norm) to implement fused batch-normalization.

```
bn = tf.contrib.layers.batch_norm(
          input_layer, fused=True, data_format='NCHW'
          scope=scope)
```

Variable Distribution and Gradient Aggregation

During training, training variable values are updated using aggregated gradients and deltas. In the benchmark script, we demonstrate that with the flexible and general-purpose TensorFlow primitives, a diverse range of high-performance distribution and aggregation schemes can be built.

Three examples of variable distribution and aggregation were included in the script:

- **`parameter_server`** where each replica of the training model reads the variables from a parameter server and updates the variable independently. When each model needs the variables, they are copied over through the standard implicit copies added by the TensorFlow runtime. The example script (https://github.com/tensorflow/benchmarks/tree/master/scripts/tf_cnn_benchmarks) illustrates using this method for local training, distributed synchronous training, and distributed asynchronous training.
- **`replicated`** places an identical copy of each training variable on each GPU. The forward and backward computation can start immediately as the variable data is immediately available. Gradients are accumulated across all GPUs, and the aggregated total is applied to each GPU's copy of the variables to keep them in sync.
- **`distributed_replicated`** places an identical copy of the training parameters on each GPU along with a master copy on the parameter servers. The forward and backward computation can start immediately as the variable data is immediately available. Gradients are accumulated across all GPUs on each server and then the per-server aggregated gradients are applied to the master copy. After all workers do this, each worker updates its copy of the variable from the master copy.

Below are additional details about each approach.

Parameter Server Variables

The most common way trainable variables are managed in TensorFlow models is parameter server mode.

In a distributed system, each worker process runs the same model, and parameter server processes own the master copies of the variables. When a worker needs a variable from a parameter server, it refers to it directly. The TensorFlow runtime adds implicit copies to the graph to make the variable value available on the computation device that needs it. When a gradient is computed on a worker, it is sent to the parameter server that owns the particular variable, and the corresponding optimizer is used to update the variable.

There are some techniques to improve throughput:

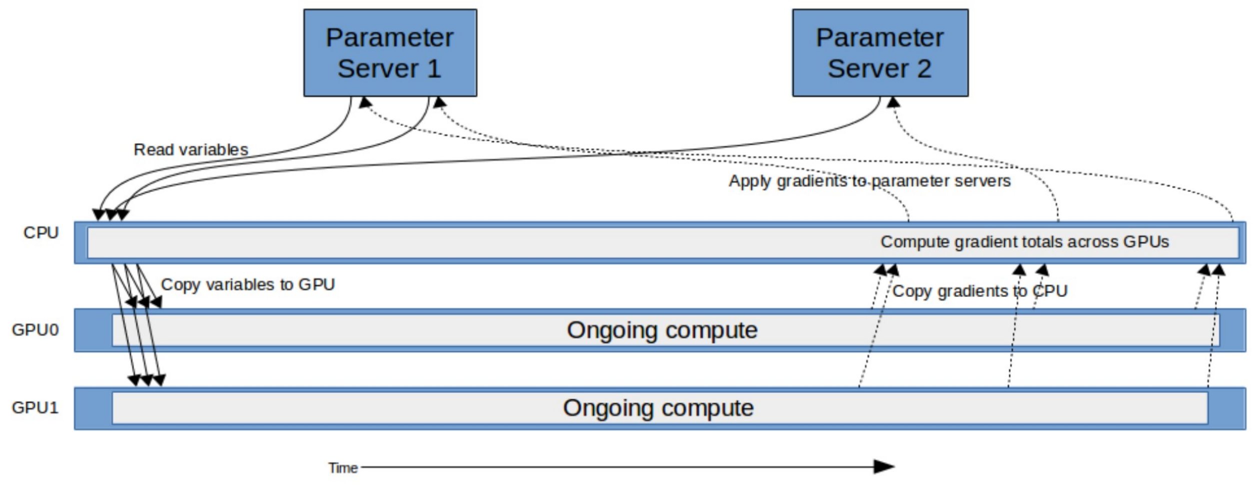
- The variables are spread among parameter servers based on their size, for load balancing.
- When each worker has multiple GPUs, gradients are accumulated across the GPUs and a single aggregated gradient is sent to the parameter server. This reduces the network bandwidth and the amount of work done by the parameter servers.

For coordinating between workers, a very common mode is async updates, where each worker updates the master copy of the variables without synchronizing with other workers. In our model, we demonstrate that it is fairly easy to introduce synchronization across workers so updates for all workers are finished in one step before the next step can start.

The parameter server method can also be used for local training, In this case, instead of spreading the master copies of variables across parameters servers, they are either on the CPU or spread across the available GPUs.

Due to the simple nature of this setup, this architecture has gained a lot of popularity within the community.

This mode can be used in the script by passing `--variable_update=parameter_server`.



Single worker's view of variable reads and updates in parameter_server mode, with three variables.

Replicated Variables

In this design, each GPU on the server has its own copy of each variable. The values are kept in sync across GPUs by applying the fully aggregated gradient to each GPU's copy of the variable.

The variables and data are available at the start of training, so the forward pass of training can start immediately. Gradients are aggregated across the devices and the fully aggregated gradient is then applied to each local copy.

Gradient aggregation across the server can be done in different ways:

- Using standard TensorFlow operations to accumulate the total on a single device (CPU or GPU) and then copy it back to all GPUs.
- Using NVIDIA® NCCL, described below in the NCCL section.

This mode can be used in the script by passing `--variable_update=replicated`.

Replicated Variables in Distributed Training

The replicated method for variables can be extended to distributed training. One way to do this like the replicated mode: aggregate the gradients fully across the cluster and apply them to each local copy of the variable. This may be shown in a future version of this scripts; the scripts do present a different variation, described here.

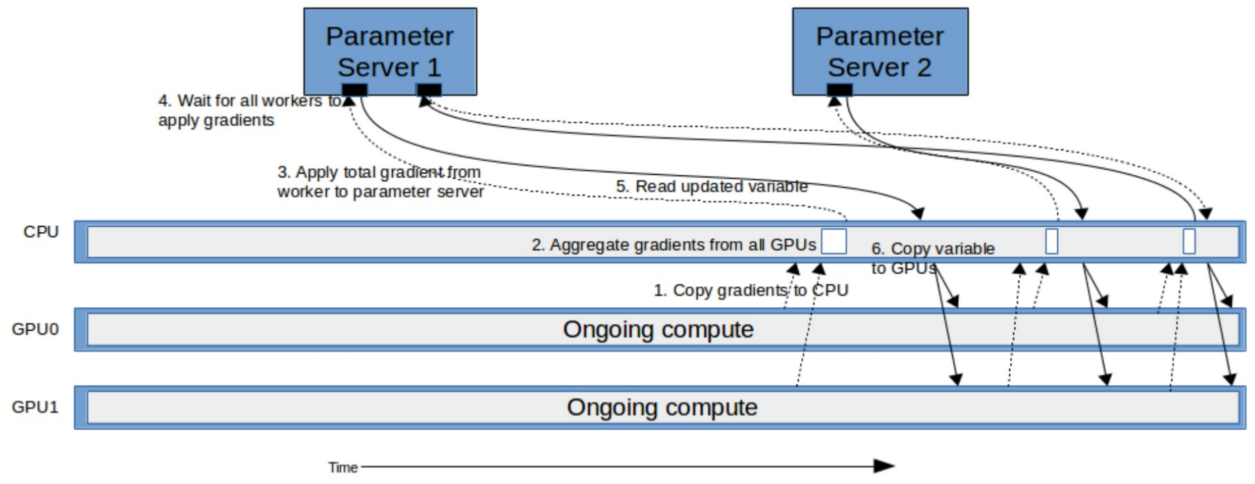
In this mode, in addition to each GPU's copy of the variables, a master copy is stored on the parameter servers. As with the replicated mode, training can start immediately using the local copies of the variables.

As the gradients of the weights become available, they are sent back to the parameter servers and all local copies are updated:

1. All the gradients from the GPU on the same worker are aggregated together.
2. Aggregated gradients from each worker are sent to the parameter server that owns the variable, where the specified optimizer is used to update the master copy of the variable.
3. Each worker updates its local copy of the variable from the master. In the example model, this is done with a cross-replica barrier that waits for all the workers to finish updating the variables, and fetches the new variable only after the barrier has been released by all replicas. Once the copy finishes for all variables, this marks the end of a training step, and a new step can start.

Although this sounds similar to the standard use of parameter servers, the performance is often better in many cases. This is largely due to the fact the computation can happen without any delay, and much of the copy latency of early gradients can be hidden by later computation layers.

This mode can be used in the script by passing `--variable_update=distributed_replicated`.



Single worker's view of variable reads and updates in distributed_replicated mode, with three variables. Ordered steps are numbered; these steps are applied for each variable.

NCCL

In order to broadcast variables and aggregate gradients across different GPUs within the same host machine, we can use the default TensorFlow implicit copy mechanism.

However, we can instead use the optional NCCL (`tf.contrib.nccl` (https://www.tensorflow.org/api_docs/python/tf/contrib/nccl)) support. NCCL is an NVIDIA® library that can efficiently broadcast and aggregate data across different GPUs. It schedules a cooperating kernel on each GPU that knows how to best utilize the underlying hardware topology; this kernel uses a single SM of the GPU.

In our experiment, we demonstrate that although NCCL often leads to much faster data aggregation by itself, it doesn't necessarily lead to faster training. Our hypothesis is that the implicit copies are essentially free since they go to the copy engine on GPU, as long as its latency can be hidden by the main computation itself. Although NCCL can transfer data faster, it takes one SM away, and adds more pressure to the underlying L2 cache. Our results show that for 8-GPUs, NCCL often leads to better performance. However, for fewer GPUs, the implicit copies often perform better.

Staged Variables

We further introduce a staged-variable mode where we use staging areas for both the variable reads, and their updates. Similar to software pipelining of the input pipeline, this can hide the data copy latency. If the computation time takes longer than the copy and aggregation, the copy itself becomes essentially free.

The downside is that all the weights read are from the previous training step. So it is a different algorithm from SGD. But it is possible to improve its convergence by adjusting learning rate and other hyperparameters.

Executing the script

This section lists the core command line arguments and a few basic examples for executing the main script (tf_cnn_benchmarks.py (https://github.com/tensorflow/benchmarks/tree/master/scripts/tf_cnn_benchmarks /tf_cnn_benchmarks.py)
).

Note: `tf_cnn_benchmarks.py` uses the config `force_gpu_compatible`, which was introduced after TensorFlow 1.1. Until TensorFlow 1.2 is released building from source is advised.

Base command line arguments

- **`model`**: Model to use, e.g. **`resnet50`**, **`inception3`**, **`vgg16`**, and **`alexnet`**.
- **`num_gpus`**: Number of GPUs to use.
- **`data_dir`**: Path to data to process. If not set, synthetic data is used. To use Imagenet data use these instructions (https://github.com/tensorflow/models/tree/master/inception#getting-started) as a starting point.
- **`batch_size`**: Batch size for each GPU.
- **`variable_update`**: The method for managing variables: **`parameter_server`** ,**`replicated`**, **`distributed_replicated`**, **`independent`**
- **`local_parameter_device`**: Device to use as parameter server: **`cpu`** or **`gpu`**.

Single instance examples

```
# VGG16 training ImageNet with 8 GPUs using arguments that optimize for
# Google Compute Engine.
python tf_cnn_benchmarks.py --local_parameter_device=cpu --num_gpus=8 \
--batch_size=32 --model=vgg16 --data_dir=/home/ubuntu/imagenet/train \
--variable_update=parameter_server --nodistortions

# VGG16 training synthetic ImageNet data with 8 GPUs using arguments that
# optimize for the NVIDIA DGX-1.
python tf_cnn_benchmarks.py --local_parameter_device=gpu --num_gpus=8 \
--batch_size=64 --model=vgg16 --variable_update=replicated --use_nccl=True

# VGG16 training ImageNet data with 8 GPUs using arguments that optimize for
# Amazon EC2.
python tf_cnn_benchmarks.py --local_parameter_device=gpu --num_gpus=8 \
--batch_size=64 --model=vgg16 --variable_update=parameter_server

# ResNet-50 training ImageNet data with 8 GPUs using arguments that optimize for
# Amazon EC2.
python tf_cnn_benchmarks.py --local_parameter_device=gpu --num_gpus=8 \
--batch_size=64 --model=resnet50 --variable_update=replicated --use_nccl=False
```

Distributed command line arguments

- **`ps_hosts`**: Comma separated list of hosts to use as parameter servers in the format of `<host>:port`, e.g. `10.0.0.2:50000`.
- **`worker_hosts`**: Comma separated list of hosts to use as workers in the format of `<host>:port`, e.g. `10.0.0.2:50001`.
- **`task_index`**: Index of the host in the list of `ps_hosts` or `worker_hosts` being started.
- **`job_name`**: Type of job, e.g `ps` or `worker`

Distributed examples

Below is an example of training ResNet-50 on 2 hosts: host_0 (10.0.0.1) and host_1 (10.0.0.2). The example uses synthetic data. To use real data pass the `--data_dir` argument.

```
# Run the following commands on host_0 (10.0.0.1):
python tf_cnn_benchmarks.py --local_parameter_device=gpu --num_gpus=8 \
--batch_size=64 --model=resnet50 --variable_update=distributed_replicated \
--job_name=worker --ps_hosts=10.0.0.1:50000,10.0.0.2:50000 \
--worker_hosts=10.0.0.1:50001,10.0.0.2:50001 --task_index=0

python tf_cnn_benchmarks.py --local_parameter_device=gpu --num_gpus=8 \
--batch_size=64 --model=resnet50 --variable_update=distributed_replicated \
--job_name=ps --ps_hosts=10.0.0.1:50000,10.0.0.2:50000 \
--worker_hosts=10.0.0.1:50001,10.0.0.2:50001 --task_index=0

# Run the following commands on host_1 (10.0.0.2):
python tf_cnn_benchmarks.py --local_parameter_device=gpu --num_gpus=8 \
--batch_size=64 --model=resnet50 --variable_update=distributed_replicated \
--job_name=worker --ps_hosts=10.0.0.1:50000,10.0.0.2:50000 \
--worker_hosts=10.0.0.1:50001,10.0.0.2:50001 --task_index=1

python tf_cnn_benchmarks.py --local_parameter_device=gpu --num_gpus=8 \
--batch_size=64 --model=resnet50 --variable_update=distributed_replicated \
--job_name=ps --ps_hosts=10.0.0.1:50000,10.0.0.2:50000 \
--worker_hosts=10.0.0.1:50001,10.0.0.2:50001 --task_index=1
```

Last updated November 2, 2017.