OPTIMIZE + DEPLOY TENSORFLOW + SPARK MODELS IN PROD W/ GPUS
STRATA LONDON, MAY 24, 2017

CHRIS FREGLY, RESEARCH ENGINEER
@ PIPELINE.IO
INTRODUCTIONS
INTRODUCTIONS: ME

- Chris Fregly, Research Engineer @ PipelineIO
  - Formerly Netflix and Databricks

- Advanced Spark and TensorFlow Meetup
  Please Join Our 15,000+ Members Globally!!

* San Francisco
* Chicago
* Washington DC
* London
ADVANCED SPARK TENSORFLOW MEETUP

Thanks, Hotels.com!

Please Join!

And Thanks, Ming L!

Video + Slides
O’REILLY ONLINE TRAINING SERIES

- High Performance TensorFlow in Production

High-Performance TensorFlow in Production
Develop hands-on experience optimizing and deploying Tensorflow models

September 7, 2017
9:00pm – 12:00am BST

SIGN UP FOR COURSE

44 spots available
Registration closes August 31, 2017 11:00 PM
INTRODUCTIONS: YOU

- **Software Engineer** or **Data Scientist** interested in optimizing and deploying TensorFlow models to production
- Assume you have a **working knowledge of TensorFlow**
CONTENT BREAKDOWN

- **50% Training Optimizations** (TensorFlow, XLA, Tools)
- **50% Deployment and Inference Optimizations** (Serving)
- Why Heavy Focus on Inference?
  - Training: boring batch, $O(\text{num}_\text{researchers})$
  - Inference: exciting realtime, $O(\text{num}_\text{users}_\text{of}_\text{app})$
- We Use Simple Models to Highlight Optimizations

**Warning:** This is not introductory TensorFlow material!
100% OPEN SOURCE CODE

- https://github.com/fluxcapacitor/pipeline/

- Please Star this Repo! 😊

- Slides, code, notebooks, Docker images available here: https://github.com/fluxcapacitor/pipeline/gpu.ml
YOU WILL LEARN...

- TensorFlow Best Practices
- To Inspect and Debug Models
- To Distribute Training Across a Cluster
- To Optimize Training with Queue Feeders
- To Optimize Training with XLA JIT Compiler
- To Optimize Inference with AOT and Graph Transform Tool (GTT)
- Key Components of TensorFlow Serving
- To Deploy Models with TensorFlow Serving
- To Optimize Inference by Tuning TensorFlow Serving
AGENDA

- GPUs and TensorFlow
- Train and Debug TensorFlow Model
- Train with Distributed TensorFlow Cluster
- Optimize Model with XLA JIT Compiler
- Optimize Model with XLA AOT and Graph Transforms
- Deploy Model to TensorFlow Serving Runtime
- Optimize TensorFlow Serving Runtime
- Wrap-up and Q&A
GPU DOCKER IMAGE

github.com/fluxcapacitor/pipeline/gpu.ml

Any username, Any password!
GPU HALF-PRECISION SUPPORT

- FP16, INT8 are “Half Precision”
- Supported by Pascal P100 (2016) and Volta V100 (2017)
- Flexible FP32 GPU Cores Can Fit 2 FP16’s for 2x Throughput!
- Half-Precision is OK for Approximate Deep Learning Use Cases

<table>
<thead>
<tr>
<th>Nvidia Tesla Workstation GPU Performance Comparison</th>
<th>P100</th>
<th>M40</th>
<th>K40</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture</td>
<td>Pascal</td>
<td>Maxwell</td>
<td>Kepler</td>
</tr>
<tr>
<td>Double Precision (FP64)</td>
<td>5.3 Tflop/s</td>
<td>0.2 Tflop/s</td>
<td>1.4 Tflop/s</td>
</tr>
<tr>
<td>Single Precision (FP32)</td>
<td>10.6 Tflop/s</td>
<td>7 Tflop/s</td>
<td>4.3 Tflop/s</td>
</tr>
<tr>
<td><strong>Half Precision (FP16)</strong></td>
<td><strong>21.1 Tflop/s</strong></td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Memory Bandwidth</td>
<td>720GB/s</td>
<td>288GB/s</td>
<td>288GB/s</td>
</tr>
<tr>
<td>Memory Size</td>
<td>16GB</td>
<td>12GB / 24GB</td>
<td>12GB</td>
</tr>
<tr>
<td>Release Date</td>
<td>2016</td>
<td>Nov-15</td>
<td>Nov-13</td>
</tr>
</tbody>
</table>
VOLTA V100 RECENTLY ANNOUNCED

- 84 Streaming Multiprocessors (SM’s)
- 5,376 GPU Cores
- 672 Tensor Cores (ie. Google TPU)
  - Mixed FP16/FP32 Precision
- More Shared Memory
- New L0 Instruction Cache
- Faster L1 Data Cache
- V100 vs. P100 Performance
  - 12x TFLOPS @ Peak Training
  - 6x Inference Throughput
Independent Thread Scheduling - Finally!!
- Similar to CPU fine-grained thread synchronization semantics
- Allows GPU to yield execution of any thread

Still Optimized for SIMT (Same Instruction Multiple Thread)
- SIMT units automatically scheduled together

Explicit Synchronization
GPU CUDA PROGRAMMING

- Barbaric, But Fun Barbaric!
- Must Know Underlying Hardware Very Well
  - Many Great Debuggers/Profilers
- Hardware Changes are Painful!
  - Newer CUDA compiler automatically JIT-compiles old CUDA code to new NVPTX
  - Not optimal, of course
CUDA STREAMS

- Asynchronous I/O Transfer
- Overlap Compute and I/O
- Keeps GPUs Saturated
- Fundamental to Queue Framework in TensorFlow
AGENDA

- GPUs and TensorFlow
- **Train and Debug TensorFlow Model**
  - Train with Distributed TensorFlow Cluster
  - Optimize Model with XLA JIT Compiler
  - Optimize Model with XLA AOT and Graph Transforms
  - Deploy Model to TensorFlow Serving Runtime
  - Optimize TensorFlow Serving Runtime
- Wrap-up and Q&A
TRAINING TERMINOLOGY

- **Tensors**: N-Dimensional Arrays
  - *e.g.* Scalar, Vector, Matrix
- **Operations**: MatMul, Add, SummaryLog, ...
- **Graph**: Graph of Operations (DAG)
- **Session**: Contains Graph(s)
- **Feeds**: Feed inputs into Operation
- **Fetches**: Fetch output from Operation
- **Variables**: What we learn through training
  - *aka* “weights”, “parameters”
- **Devices**: Hardware device on which we train

```python
with tf.device("worker:0/device/gpu:0, worker:1/device/gpu:0")
```
TRAINING DEVICES

- cpu: 0
  - By default, all CPUs
  - Requires extra config to target a CPU
- gpu: 0..n
  - Each GPU has a unique id
  - TF usually prefers a single GPU
- xla_cpu: 0, xla_gpu: 0..n
  - “JIT Compiler Device”
  - Hints TF to attempt JIT Compile
**TRAINING METRICS: TENSORBOARD**

- **Summary Ops**
  
  ```python
  loss_summary_op = tf.summary.scalar('loss', loss)
  merge_all_summary_op = tf.summary.merge_all()
  summary_writer = tf.summary.FileWriter('/root/tensorboard/linear/<version>', graph=sess.graph)
  ```

- **Event Files**
  
  `/root/tensorboard/linear/<version>/events`

- **Tags**
  
  - Organize data within Tensorboard UI
TRAINING ON EXISTING INFRASTRUCTURE

- Data Processing
  - HDFS/Hadoop
  - Spark

- Containers
  - Docker

- Schedulers
  - Kubernetes
  - Mesos

https://github.com/tensorflow/ecosystem

<dependency>
  <groupId>org.tensorflow</groupId>
  <artifactId>tensorflow-hadoop</artifactId>
  <version>1.0-SNAPSHOT</version>
</dependency>
FEED TRAINING DATA TO TENSORFLOW

- Don’t Use `feed_dict` for Production Workloads!!
  - `feed_dict` Requires C++ <-> Python Serialization
  - Batch Retrieval is Single-threaded, Synchronous, SLOW!
  - Next Batch Not Retrieved Until Current Batch is Complete
  - CPUs and GPUs are Not Fully Utilized!
- Solution: Use Queues to Read and Pre-Process Batches
  - Queues perform I/O, pre-processing, shuffling, ...
  - Queues should use CPUs to keep GPU focused on compute
DATA MOVEMENT WITH QUEUES

- Queue Pulls Batch from Source (ie HDFS, Kafka)
- Queue Pre-Process Data (Usually CPUs Only)
  - Use ShuffleQueue to create stochastic mini-batches
  - Combine many small files into a few large TFRecord files
- GPU Pulls Batch from Queue (CUDA Streams)
  - GPU pulls next batch while processing current batch

GPUs Fully Utilized!
QUEUE CAPACITY PLANNING

- **batch_size**
  - # of examples per batch (ie. 64 jpg)
  - Limited by GPU RAM

- **num_processing_threads**
  - CPU threads pull and pre-process batches of data
  - Limited by CPU Cores

- **queue_capacity**
  - Limited by CPU RAM (ie. 5 * batch_size)

---

Saturate those GPUs!

GPU Pulls Batches while Processing Current Batch

Async Memory Transfer with CUDA Streams

-- Thanks, Nvidia!! --
DETECT UNDERUTILIZED CPUS, GPUS

• Instrument training code to generate "timelines"

```python
from tensorflow.python.client import timeline

trace = timeline.Timeline(step_stats=run_metadata.step_stats)

with open('timeline.json', 'w') as trace_file:
    trace_file.write(
        trace.generate_chrome_trace_format(show_memory=True))
```

• Analyze with Google Web Tracing Framework (WTF)

http://google.github.io/tracing-framework/

• Monitor CPU with `top`, GPU with `nvidia-smi`
TENSORFLOW MODEL

- MetaGraph
  - Combines GraphDef and Metadata
- GraphDef
  - Architecture of your model (nodes, edges)
- Metadata
  - Asset: Accompanying assets to your model
  - SignatureDef: Maps external -> internal tensors
- Variables
  - Stored separately during training (checkpoint)
  - Allows training to continue from any checkpoint
  - Variables are “frozen” into Constants when deployed for inference

Variables:
“W” : 0.328
“b” : -1.407
TENSORFLOW SESSION

Session

\[
\text{graph: GraphDef}
\]

Variables:

\[
\begin{align*}
\text{"W"} & : \ 0.328 \\
\text{"b"} & : \ -1.407
\end{align*}
\]
TENSORFLOW DEBUGGER

- Step through Operations
- Inspect Inputs and Outputs
- Wrap Session in Debug Session

```
sess = tf.Session(config=config)
sess = tf_debug.LocalCLIDebugWrapperSession(sess)
```
AGENDA

- GPUs and TensorFlow
- Train and Debug TensorFlow Model
- Train with Distributed TensorFlow Cluster
- Optimize Model with XLA JIT Compiler
- Optimize Model with XLA AOT and Graph Transforms
- Deploy Model to TensorFlow Serving Runtime
- Optimize TensorFlow Serving Runtime
- Wrap-up and Q&A
MULTI-GPU TRAINING (SINGLE NODE)

- Variables stored on CPU (cpu: 0)
- Model graph (aka “replica”, “tower”) is copied to each GPU (gpu: 0, gpu: 1, …)

Multi-GPU Training Steps:
1. CPU transfers model to each GPU
2. CPU waits on all GPUs to finish batch
3. CPU copies all gradients back from all GPUs
4. CPU synchronizes and averages all gradients from GPUs
5. CPU updates GPUs with new variables/weights
6. Repeat Step 1 until reaching stop condition (ie. max_epochs)
DISTRIBUTED, MULTI-NODE TRAINING

- TensorFlow Automatically Inserts Send and Receive Ops into Graph
- Parameter Server Synchronously Aggregates Updates to Variables
- Nodes with Multiple GPUs will Pre-Aggregate Before Sending to PS
SYNCHRONOUS VS. ASYNCHRONOUS

- Synchronous
  - Worker ("graph replica", "tower")
    - Reads same variables from Parameter Server in parallel
    - Computes gradients for variables using partition of data
    - Sends gradients to central Parameter Server
  - Parameter Server
    - Aggregates (avg) gradients for each variable based on its portion of data
    - Applies gradients (+, -) to each variables
    - Broadcasts updated variables to each node in parallel
    - ^^^ Repeat ^^^
- Asynchronous
  - Each node computes gradients independently
  - Reads stale values, does not synchronized with other nodes
DATA PARALLEL VS MODEL PARALLEL

- **Data Parallel ("Between-Graph Replication")**
  - Send **exact same model** to each device
  - Each device operates on its **partition of data**
    - i.e. Spark sends same function to many workers
    - Each worker operates on their partition of data

- **Model Parallel ("In-Graph Replication")**
  - Send **different partition of model** to each device
  - Each device operates on **all data**

*Very Difficult!!*

*Required for Large Models.*

*(GPU RAM Limitation)*
DISTRIBUTED TENSORFLOW CONCEPTS

- **Client**
  - Program that builds a TF Graph, constructs a session, interacts with the cluster
  - Written in Python, C++

- **Cluster**
  - Set of distributed nodes executing a graph
  - Nodes can play any role

- **Jobs ("Roles")**
  - Parameter Server ("ps") stores and updates variables
  - Worker ("worker") performs compute-intensive tasks (stateless)
  - Assigned 0..* tasks

- **Task ("Server Process")**

---

“ps” and “worker” are named by convention
CHIEF WORKER

- Worker Task 0 is Chosen by Default
  - Task 0 is guaranteed to exist
- Implements Maintenance Tasks
  - Writes checkpoints
  - initializes parameters at start of training
  - Writes log summaries
  - Parameter Server health checks

```python
with tf.Session(server.target) as sess:
    while True:
        # ...
        One worker task acts as "chief"
        if is_chief and step % 1000 == 0:
            saver.save(sess, "/home/mrry/..."
```
NODE AND PROCESS FAILURES

- Checkpoint to Persistent Storage (HDFS, S3)
- Use MonitoredTrainingSession and Hooks
- Use a Good Cluster Orchestrator (i.e., Kubernetes, Mesos)
- Understand Failure Modes and Recovery States

Stateless, Not Bad: Training Continues

Stateful, Bad: Training Must Stop

Dios Mio! Long Night Ahead...
SHARDED SAVERS

- `tf.train.Saver(sharded=True)`
- Allows Each PS to Persist Independently
- Otherwise, All Vars from All PS’s Collected on 1 PS
  - Hello, OOM Error!
VALIDATING DISTRIBUTED MODEL

- Use Separate Scorer Cluster to Avoid Resource Contention
- Validate using Saved Checkpoints from Parameter Servers
EXPERIMENT AND ESTIMATOR API

- Higher-Level APIs Simplify Distributed Training
- Picks Up Configuration from Environment
- Supports Custom Models (ie. Keras)
- Used for Training, Validation, and Prediction
- API is Changing, but Patterns Remain the Same
- Works Well with Google Cloud ML (Surprised?!)
AGENDA

- GPUs and TensorFlow
- Train and Debug TensorFlow Model
- Train with Distributed TensorFlow Cluster
  - Optimize Model with XLA JIT Compiler
  - Optimize Model with XLA AOT and Graph Transforms
- Deploy Model to TensorFlow Serving Runtime
- Optimize TensorFlow Serving Runtime
- Wrap-up and Q&A
XLA FRAMEWORK

- Accelerated Linear Algebra (XLA)
- Goals:
  - Reduce reliance on custom operators
  - Improve execution speed
  - Improve memory usage
  - Reduce mobile footprint
  - Improve portability
- Helps TF Stay Flexible and Performant
XLA HIGH LEVEL OPTIMIZER (HLO)

- Compiler Intermediate Representation (IR)
- Independent of source and target language
- Define Graphs using HLO Language
- XLA Step 1 Emits Target-Independent HLO
- XLA Step 2 Emits Target-Dependent LLVM
- LLVM Emits Native Code Specific to Target
- Supports x86-64, ARM64 (CPU), and NVPTX (GPU)
JIT COMPILER

- Just-In-Time Compiler
- Built on XLA Framework
- Goals:
  - Reduce memory movement – especially useful on GPUs
  - Reduce overhead of multiple function calls
  - Similar to Spark Operator Fusing in Spark 2.0
  - Unroll Loops, Fuse Operators, Fold Constants, ...
  - Scope to session, device, or `with `jit_scope():`
VISUALIZING JIT COMPILER IN ACTION

Before

After

Google Web Tracing Framework:
http://google.github.io/tracing-framework/

from tensorflow.python.client import timeline
trace = timeline.Timeline(step_stats=run_metadata.step_stats)
with open('timeline.json', 'w') as trace_file:
    trace_file.write(trace.generate_chrome_trace_format(show_memory=True))
VISUALIZING FUSING OPERATORS

GraphViz:
http://www.graphviz.org

pip install graphviz

dot -Tpng \
/tmp/hlo_graph_99.w5LcGs.dot \n-o hlo_graph_80.png

hlo_*.dot files generated by XLA
IT’S WORTH HIGHLIGHTING…

- From Now On, We Optimize Trained Models For Inference
- In Other Words,

We’re Done with Training! Yeah!!
AGENDA

- GPUs and TensorFlow
- Train and Debug TensorFlow Model
- Train with Distributed TensorFlow Cluster
- Optimize Model with XLA JIT Compiler
- Optimize Model with XLA AOT and Graph Transforms
- Deploy Model to TensorFlow Serving Runtime
- Optimize TensorFlow Serving Runtime
- Wrap-up and Q&A
AOT COMPILER

- Standalone, Ahead-Of-Time (AOT) Compiler
- Built on XLA framework
- `tfcompile`
- Creates executable with minimal TensorFlow Runtime needed
  - Includes only dependencies needed by subgraph computation
- Creates functions with `feeds` (inputs) and `fetches` (outputs)
  - Packaged as `cc_library` header and object files to link into your app
- Commonly used for mobile device inference graph
  - Currently, only CPU x86-64 and ARM are supported - no GPU
GRAPH TRANSFORM TOOL (GTT)

- Optimize Trained Models for Inference
- Remove training-only Ops (checkpoint, drop out, logs)
- Remove unreachable nodes between given feed -> fetch
- Fuse adjacent operators to improve memory bandwidth
- Fold final batch norm mean and variance into variables
- Round weights/variables improves compression (ie. 70%)
- Quantize weights and activations simplifies model
  - FP32 down to INT8
BEFORE OPTIMIZATIONS
AFTER STRIPPING UNUSED NODES

- Optimizations
  - strip_unused_nodes

- Results
  - Graph much simpler
  - File size much smaller
AFTER REMOVING UNUSED NODES

- Optimizations
  - strip_unused_nodes
  - remove_nodes
- Results
  - Pesky nodes removed
  - File size a bit smaller
AFTER FOLDING CONSTANTS

- Optimizations
  - strip_unused_nodes
  - remove_nodes
  - fold_constants
- Results
  - $w$ and $b$ become variables, not placeholders (feeds)
What is Batch Normalization?
- Each batch of data may have wildly different distributions
- Normalize per batch (and layer)
- Speeds up training dramatically
- Weights are learned quicker
- Final model is more accurate

**Always Use Batch Normalization!**

GTT Fuses Final mean and variance MatMul into Graph
AFTER FOLDING BATCH NORMS

- Optimizations
  - strip_unused_nodes
  - remove_nodes
  - fold_constants
  - fold_batch_norms
- Results
  - Graph remains the same, file size approximately the same
WEIGHT QUANTIZATION

- FP16 and INT8 Are Smaller and Computationally Simpler
- Weights/Variables are Constants
- Easy to Linearly Quantize
AFTER QUANTIZING WEIGHTS

- Optimizations
  - strip_unused_nodes
  - remove_nodes
  - fold_constants
  - fold_batch_norms
  - quantize_weights

- Results
  - Graph remains the same, file size is smaller
**ACTIVATION QUANTIZATION**

- Activations Not Known Ahead of Time
  - Depends on input, not easy to quantize
- Requires Calibration Step
  - Use a “representative” dataset
- Per Neural Network Layer...
  - Collect histogram of activation values
  - Generate many quantized distributions with different saturation thresholds
  - Choose threshold to minimize...
    - $KL_{\text{divergence}}(\text{ref\_distribution}, \text{quant\_distribution})$
- Not Much Time or Data is Required (Minutes on Commodity Hardware)
ACTIVATION QUANTIZATION GRAPH OPS

Create Conversion Subgraph

Produces QuantizedMatMul, QuantizedRelu

Eliminate Adjacent Dequantize + Quantize
AFTER QUANTIZING ACTIVATIONS

- Optimizations
  - strip_unused_nodes
  - remove_nodes
  - fold_constants
  - fold_batch_norms
  - quantize_weights
  - quantize_nodes (activations)

- Results
  - Larger graph, needs calibration!
AGENDA

- GPUs and TensorFlow
- Train and Debug TensorFlow Model
- Train with Distributed TensorFlow Cluster
- Optimize Model with XLA JIT Compiler
- Optimize Model with XLA AOT and Graph Transforms
- **Deploy Model to TensorFlow Serving Runtime**
- Optimize TensorFlow Serving Runtime
- Wrap-up and Q&A
MODEL SERVING TERMINOLOGY

- **Inference**
  - Only Forward Propagation through Network
  - Predict, Classify, Regress, …

- **Bundle**
  - GraphDef, Variables, Metadata, …

- **Assets**
  - ie. Map of ClassificationID -> String
    - {9283: “penguin”, 9284: “bridge”, …}

- **Version**
  - Every Model Has a Version Number (Integers Only?!)

- **Version Policy**
  - ie. Serve Only Latest (Highest), Serve both Latest and Previous, …
TENSORFLOW SERVING FEATURES

- Low-latency or High-throughput Tuning
- Supports Auto-Scaling
- Different Models/Versions Served in Same Process
- Custom Loaders beyond File-based
- Custom Serving Models beyond HashMap and TensorFlow
- Custom Version Policies for A/B and Bandit Tests
- Drain Requests for Graceful Model Shutdown or Update
- Extensible Request Batching Strategies for Diff Use Cases and HW
- Uses Highly-Efficient GRPC and Protocol Buffers
PREDICTION SERVICE

- **Predict (Original, Generic)**
  - Input: List of Tensors
  - Output: List of Tensors

- **Classify**
  - Input: List of `tf.Example` (key, value) pairs
  - Output: List of `(class_label: String, score: float)`

- **Regress**
  - Input: List of `tf.Example` (key, value) pairs
  - Output: List of `(label: String, score: float)`
PREDICTION INPUTS + OUTPUTS

- **SignatureDef**
  - Defines inputs and outputs
  - Maps external (logical) to internal (physical) tensor names
  - Allows internal (physical) tensor names to change

```python
# Define tensor_info_x_observed
tensor_info_x_observed = utils.build_tensor_info(x_observed)

# Define tensor_info_y_pred
tensor_info_y_pred = utils.build_tensor_info(y_pred)

# Define prediction_signature
prediction_signature = signature_def_utils.build_signature_def(
    inputs = {'x_observed': tensor_info_x_observed},
    outputs = {'y_pred': tensor_info_y_pred},
    method_name = signature_constants.PREDICT_METHOD_NAME
)
```
MULTI-HEADED INFRINGEMENT

- Multiple “Heads” of Model
- Return class and scores to be fed into another model
- Inputs Propagated Forward Only Once
- Optimizes Bandwidth, CPU, Latency, Memory, Coolness
BUILD YOUR OWN MODEL SERVER (?!)

- Adapt GRPC (Google) <-> HTTP (REST of the World)
- Perform Batch Inference vs. Request/Response
- Handle Requests Asynchronously
- Support Mobile, Embedded Inference
- Customize Request Batching
- Add Circuit Breakers, Fallbacks
- Control Latency Requirements
- Reduce Number of Moving Parts

```cpp
#include "tensorflow_serving/model_servers/server_core.h"
...

class MyTensorFlowModelServer {
  ServerCore::Options options;
  // set options (model name, path, etc)
  std::unique_ptr<ServerCore> core;

  TF_CHECK_OK(
    ServerCore::Create(std::move(options), &core);
  )
};
```

Compile and Link with `libtensorflow.so`
AGENDA

- GPUs and TensorFlow
- Train and Debug TensorFlow Model
- Train with Distributed TensorFlow Cluster
- Optimize Model with XLA JIT Compiler
- Optimize Model with XLA AOT and Graph Transforms
- Deploy Model to TensorFlow Serving Runtime
- Optimize TensorFlow Serving Runtime
- Wrap-up and Q&A
REQUEST BATCH TUNING

- **max_batch_size**
  - Enables throughput/latency tradeoff
  - Bounded by RAM

- **batch_timeout_micros**
  - Defines batch time window, latency upper-bound
  - Bounded by RAM

- **num_batch_threads**
  - Defines parallelism
  - Bounded by CPU cores

- **max_enqueued_batches**
  - Defines queue upper bound, throttling
  - Bounded by RAM

Reaching either threshold will trigger a batch.
**Batch Scheduler Strategies**

- **BasicBatchScheduler**
  - Best for homogeneous request types (i.e., always classify or always regress)
  - Async callback when `max_batch_size` or `batch_timeout_micros` is reached
  - `BatchTask` encapsulates unit of work to be batched

- **SharedBatchScheduler**
  - Best for heterogeneous request types, multi-step inference, ensembles, ...
  - Groups BatchTasks into separate queues to form homogenous batches
  - Processes batches fairly through interleaving

- **StreamingBatchScheduler**
  - Mixed CPU/GPU/IO-bound workloads
  - Provides fine-grained control for complex, multi-phase inference logic

*Must Experiment to Find the Best Strategy for You!!*
AGENDA

- GPUs and TensorFlow
- Train and Debug TensorFlow Model
- Train with Distributed TensorFlow Cluster
- Optimize Model with XLA JIT Compiler
- Optimize Model with XLA AOT and Graph Transforms
- Deploy Model to TensorFlow Serving Runtime
- Optimize TensorFlow Serving Runtime

- Wrap-up and Q&A
YOU JUST LEARNED...

- TensorFlow Best Practices
- To Inspect and Debug Models
- To Distribute Training Across a Cluster
- To Optimize Training with Queue Feeders
- To Optimize Training with XLA JIT Compiler
- To Optimize Inference with AOT and Graph Transform Tool (GTT)
- Key Components of TensorFlow Serving
- To Deploy Models with TensorFlow Serving
- To Optimize Inference by Tuning TensorFlow Serving
Q&A

- Thank you!!

- https://github.com/fluxcapacitor/pipeline/

- Slides, code, notebooks, Docker images available here: https://github.com/fluxcapacitor/pipeline/gpu.ml

Contact Me @

Email: chris@pipeline.io
Twitter: @cfregly