GPU OPTIMISATION FOR CPATH

Jonny Hancox
jhancox@nvidia.com
AGENDA

- GPUs - Why and How?
- 5 Optimisation Tips
- Questions
EVOLUTION OF COMPUTING

1995
PC Internet
WinTel, Yahoo!
1 billion PC users

2005
Mobile-Cloud
iPhone, Amazon AWS
2.5 billion mobile users

2015
AI & IOT
Deep Learning, GPU
100s of billions of devices
RISE OF GPU COMPUTING

Alex Krishevsky’s ‘AlexNet’ spawned a new generation of GPU-powered DNNs.
HOW GPU ACCELERATION WORKS

Application Code

Compute-Intensive Functions
5% of Code

Rest of Sequential CPU Code

GPU

CPU
ACCELERATED COMPUTING
10X PERFORMANCE & 5X ENERGY EFFICIENCY FOR HPC

<table>
<thead>
<tr>
<th>GPU Accelerator</th>
<th>Optimized for Parallel Tasks</th>
</tr>
</thead>
</table>

**ACCELERATED COMPUTING**

**CPU Optimized for Serial Tasks**

**GPU Accelerator**

**Optimized for Parallel Tasks**

---

**GPU Strengths**

- High bandwidth main memory
- Significantly more compute resources
- Latency tolerant via parallelism
- High throughput
- High performance/watt
- No context-switching cost

**GPU Weaknesses**

- Relatively low memory capacity
- Low per-thread performance
TIP #1 - USE OPTIMISED SOFTWARE
GPU ACCELERATED LIBRARIES

“Drop-in” Acceleration for Your Applications

DEEP LEARNING
- cuDNN
- TensorRT
- DeepStream SDK

SIGNAL, IMAGE & VIDEO
- cuFFT
- NVIDIA NPP
- CODEC SDK

LINEAR ALGEBRA
- cuBLAS
- CUDA Math library
- cuSOLVER
- cuRAND

PARALLEL ALGORITHMS
- nvGRAPH
- NCCL
- Thrust

CUDA Math library
- cuDNN
- cuSOLVER
- cuRAND

DeepStream SDK
- NVIDIA NPP
- CODEC SDK
CUDA-X AI TRANSFORMS DATA SCIENCE

From Data Science to NVIDIA Accelerated Data Science with CUDA-X AI
NVIDIA GPU CLOUD
https://ngc.nvidia.com

Simple Access to a Comprehensive Catalog of GPU-accelerated Software

Discover 50+ GPU-Accelerated Containers
Deep learning, third-party managed HPC applications, NVIDIA HPC visualization tools, and partner applications

Innovate in Minutes, Not Weeks
Get up and running quickly and reduce complexity

Access from Anywhere
Use on PCs with NVIDIA Volta or Pascal™ architecture GPUs, NVIDIA DGX Systems, and supported cloud providers
GET STARTED WITH NGC

Explore the NGC Registry for DL, ML & HPC

Deploy containers: ngc.nvidia.com

Learn more about NGC offering: nvidia.com/ngc

Technical information: developer.nvidia.com
NGC: GPU-OPTIMIZED SOFTWARE HUB

Simplifying DL, ML and HPC Workflows

50+ Containers
DL, ML, HPC

15+ Model Training Scripts
NLP, Image Classification, Object Detection & more

60 Pre-trained Models
NLP, Image Classification, Object Detection & more

Industry Workflows
Medical Imaging, Intelligent Video Analytics

DEEP LEARNING
TensorFlow | PyTorch | more

MACHINE LEARNING
RAPIDS | H2O | more

HPC
NAMD | GROMACS | more

VISUALIZATION
ParaView | IndeX | more
TIP #2  - PROFILE YOUR CODE
“Premature optimization is the root of all evil”

Donald Knuth
Training Deep Learning Models is a time and compute intensive process. Optimizing Deep Learning Models involves iterating through:

- Locating Optimization Opportunities (Profiling)
- Visualizing and analyzing Key areas of improvement
- Taking Action

Profiling helps to find where to update code to accelerate training time on GPUs without loss of accuracy.
PROFILING TOOLS & TECHNOLOGIES

Researchers

- NVTX
- Nsight Systems
- Nsight Compute

Data Scientists & Applied Researchers

- DLProf
- Tensorboard
- <Nsight Systems w/ NVTX>

Sysadmins & DevOps

- Data Center Monitoring Tools
- DCGM, NVML
- <Nsight Systems>

Skills in Algorithms ------------------- Skills in Domains & Applications--------------- Skills in Systems
DATA SCIENTIST & RESEARCHER CHALLENGES

- Limited CUDA skills
- Using Tensorboard but no GPU time or usage
- How to locate opportunities for speed ups using reduced precision
- Reason why some operations are not using Tensor cores could be as easy as dimensions of the matrix are not optimal

How do we discover where performance can be improved?
NVIDIA PROFILING TOOLS

Deep Learning Profiler (DLProf)

Nsight Systems  Nsight Compute

NVTX for Tensorflow  NVTX for PyTorch  NVTX for MXNet
SIMPLE TRAINING PROGRAM

• A simple DNN training program
• A GPU accelerated pipeline
• Training is done in batches and epochs
  1. Data is copied to the device
  2. Forward pass
  3. Backward pass
def train(args, model, device, train_loader, optimizer, epoch):
    model.train()
    for batch_idx, (data, target) in enumerate(train_loader):
        data, target = data.to(device), target.to(device)
        optimizer.zero_grad()
        output = model(data)
        loss = F.nll_loss(output, target)
        loss.backward()
        optimizer.step()
        if batch_idx % args.log_interval == 0:
            print('Train Epoch: {} [{}/{} ({:.0f}%)]
                  Loss: {:.6f}'.format(
                epoch, batch_idx * len(data), len(train_loader.dataset),
                100. * batch_idx / len(train_loader), loss.item()))
TRAINING TIME

- We ran the training program:
  -> python main.py

- It took 89 seconds to complete!
NVIDIA NSIGHT SYSTEMS

System Wide Profiling Tool

- Balance your workload across multiple CPUs and GPUs
- Locate idle CPU and GPU time
- Locate redundant synchronizations
- Locate optimization opportunities
- Improve application’s performance
LAUNCH NSIGHT SYSTEMS ON TARGET MACHINE

- `nsys profile -y 10 -d 15 -w true -t "cudnn,cuda,osrt,nvtx" -o ~/data/session1 python main.py`

- **profile** - start a profiling session
- -y - collection start delay in seconds
- -d - collection duration in seconds
- -w true - send target process’ stdout and stderr streams to the console
- -t - selects the APIs to be traced
- -o - name for the intermediate result file, created at the end of the collection
PROFILING

- Training time is 89 seconds
- The CPU waits on a semaphore and starves the GPU!
NVTX PERFORMANCE MARKERS

NVTX performance markers annotate the timeline with application’s logic

def train(args, model, device, train_loader, optimizer, epoch):
    model.train()
    for batch_idx, (data, target) in enumerate(train_loader):
        nvtx.range_push("Batch " + str(batch_idx))

        nvtx.range_push("Copy to device")
        data, target = data.to(device), target.to(device)
        nvtx.range_pop()

        nvtx.range_push("Forward pass")
        optimizer.zero_grad()
        output = model(data)
        loss = F.nll_loss(output, target)
        nvtx.range_pop()
PROFILING WITH NVTX

- We see that the GPU is idle when data is loaded.
- Data is loaded using a single thread. This starves the GPU!
EXAMPLE TUNING

Looking at the source code reveals the problem:

```python
kwargs = {'num_workers': 1, 'pin_memory': True} if use_cuda else {}
```

Let’s switch to using 8 working threads:

```python
kwargs = {'num_workers': 8, 'pin_memory': True} if use_cuda else {}
```
We now see that data is loaded using 8 threads.
Training time dropped from 89 to 21 seconds!
When using TensorFlow 1.x, the python program builds the computation graph. Then the graph is executed using C++ code.

- **nvtx_plugins**
  - Enables wrapping regions of the computation graph with `nvtx start` and `end` operations.
  - Also provides Keras callbacks and session hooks.
import nvtx.plugins.tf as nvtx_tf

x, nvtx_context = nvtx_tf.ops.start(x, message='Dense 1-3',
                      domain_name='Forward', grad_domain_name='Gradient')
x = tf.layers.dense(x, 1000, activation=tf.nn.relu, name='dense_1')
x = tf.layers.dense(x, 1000, activation=tf.nn.relu, name='dense_2')
x = tf.layers.dense(x, 1000, activation=tf.nn.relu, name='dense_3')
x = nvtx_tf.ops.end(x, nvtx_context)
x = tf.layers.dense(x, 1000, activation=tf.nn.relu, name='dense_4')
DLPROF

Adds profiler data into TensorBoard

<table>
<thead>
<tr>
<th>GPU Time (ms)</th>
<th>CPU Time (ms)</th>
<th>Op Name</th>
<th>Op Type</th>
<th>Graph</th>
<th>Calls</th>
<th>TC Flatt</th>
<th>TC Virtual</th>
</tr>
</thead>
<tbody>
<tr>
<td>23791.6</td>
<td>33871.1</td>
<td>gradient/gradient/...</td>
<td>Core2D</td>
<td>GraphEd</td>
<td>100</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>859210</td>
<td>1023456</td>
<td>gradient/gradient/...</td>
<td>Core2D</td>
<td>GraphEd</td>
<td>100</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>123456</td>
<td>123456</td>
<td>gradient/gradient/...</td>
<td>Core2D</td>
<td>GraphEd</td>
<td>100</td>
<td>true</td>
<td>true</td>
</tr>
</tbody>
</table>

Kernel Summaries

<table>
<thead>
<tr>
<th>Kernel Name</th>
<th>Using TC</th>
<th>Calls</th>
<th>GPU Time (ms)</th>
<th>Arg0 (ms)</th>
<th>Mib0 (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>void ...</td>
<td>false</td>
<td>103</td>
<td>12357</td>
<td>1267</td>
<td>1284</td>
</tr>
<tr>
<td>void ...</td>
<td>false</td>
<td>103</td>
<td>125870</td>
<td>1174</td>
<td>1167</td>
</tr>
<tr>
<td>void ...</td>
<td>false</td>
<td>103</td>
<td>212</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>
$ dlprof --in_graphdef=graphdef.pbtxt /usr/bin/python nvidia-examples/cnn/resnet.py --layers=50 --num_iter=100 --iter_unit=batch --display_every=50 --data_dir=/data/train-images --batch=128 --precision=fp16

Use NVIDIA Optimized Tensorflow Framework container

- **Generate graph definition**
- **Prefix training script with dlprof**
- **Visualize with Tensorboard or text reports**

VISUALISATION

NVIDIA Modifications to Tensorboard to reflect GPU details

- Puts a **GPU Lens** on Tensorboard
- Tensor Core Compatibility
  - Eligible operations that could use Tensor cores
- Drill down to examine deeper on the green colored nodes
  - Ops that used Tensor cores
  - Ops that did not
TIP #3 - USE MIXED PRECISION
TENSOR CORES & MIXED PRECISION

FP16 storage/input
Full precision product
Sum with FP32 accumulator

F16  X  +  F32

F32

F16

Models
CNN, RNN, GAN, RL, NCF...

Frameworks
NVIDIA AMP
Tensor Cores
FP32 AND FP16

**FP32**

8-bit exponent, 23-bit mantissa

- Dynamic range: $1.4 \times 10^{-45} < x < 3.4 \times 10^{38}$

**FP16**

5-bit exponent, 10-bit mantissa

- Dynamic range: $5.96 \times 10^{-8} < x < 65504$
MIXED PRECISION TRAINING

Motivation

• Balance a pure tradeoff of speed and accuracy:
  • Reduced precision (16-bit floating point) for speed or scale
  • Full precision (32-bit floating point) to maintain task-specific accuracy

• Under the constraints:
  • Maximize use of reduced precision while matching accuracy of full precision training
  • No changes to hyperparameters

MAXIMIZING MODEL PERFORMANCE

FP16 is fast and memory-efficient.

FP32

1x compute throughput
1x memory throughput
1x memory storage

FP16

8X compute throughput
2X memory throughput
1/2X memory storage
AUTOMATIC MIXED PRECISION

Speedup Your Network Across Frameworks With Just Two Lines of Code

“This easy integration enables TensorFlow developers to literally flip a switch in their AI model and get up to 3X speedup with mixed precision training while maintaining model accuracy.”

Rajat Monga, Engineering Director, TensorFlow
MIXED PRECISION TRAINING

With Tensor Cores

- 8GPU training of ResNet-50 (ImageNet classification) on DGX-1
  - NVIDIA mxnet-18.08-py3 container
- Total time to run full training schedule in mixed precision is well under four hours
  - 2.9x speedup over FP32 training
  - Equal validation accuracies
  - No hyperparameters changed
    - Minibatch = 256 per GPU
## MIXED PRECISION IS GENERAL PURPOSE

Models trained to match FP32 results (same hyperparameters)

<table>
<thead>
<tr>
<th>Image Classification</th>
<th>Detection / Segmentation</th>
<th>Generative Models (Images)</th>
<th>Language Modeling</th>
</tr>
</thead>
<tbody>
<tr>
<td>AlexNet</td>
<td>DeepLab</td>
<td>DLSS</td>
<td>BERT</td>
</tr>
<tr>
<td>DenseNet</td>
<td>Faster R-CNN</td>
<td>Partial Image Inpainting</td>
<td>BigLSTM</td>
</tr>
<tr>
<td>Inception</td>
<td>Mask R-CNN</td>
<td>Progress GAN</td>
<td>8k mLSTM (NVIDIA)</td>
</tr>
<tr>
<td>MobileNet</td>
<td>Multibox SSD</td>
<td>Pix2Pix</td>
<td>Translation</td>
</tr>
<tr>
<td>NASNet</td>
<td>NVIDIA Automotive</td>
<td>Speech</td>
<td>FairSeq (convolution)</td>
</tr>
<tr>
<td>ResNet</td>
<td>RetinaNet</td>
<td>Deep Speech 2</td>
<td>GNMT (RNN)</td>
</tr>
<tr>
<td>ResNeXt</td>
<td>UNET</td>
<td>Tacotron</td>
<td>Transformer (self-attention)</td>
</tr>
<tr>
<td>VGG</td>
<td></td>
<td>WaveNet</td>
<td></td>
</tr>
<tr>
<td>XCeption</td>
<td></td>
<td>WaveGlow</td>
<td></td>
</tr>
<tr>
<td>Recommendation</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DeepRecommender</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NCF</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
MIXED PRECISION TRAINING

Key Observations

- FP16 dynamic range is sufficient for training, but gradients may have to be scaled to move them into the range to keep them from becoming zeros in FP16.

- Main idea: Choose a constant scaling factor $S$ so that its product with the maximum absolute gradient value is below 65,504 (the maximum value representable in FP16).

- Problem: If no gradient statistics are available, overflow may occur when choosing $S$ too large. $\rightarrow$ Dynamic Loss Scaling

- Some networks require updating an FP32 copy of weights. Also, values computed by large reductions should be left in FP32, e.g. mean and variance in batch-normalization.
MIXED PRECISION TRAINING

Three-part methodology

- **Model conversion:**
  - Switch everything to run on FP16 values
  - Insert casts to FP32 for loss functions and normalization/pointwise ops that need full precision

- **Master weights:**
  - Keep FP32 model parameters, update at each iteration
  - Use an FP16-casted copy for both forward pass and backpropagation

- **Loss scaling:**
  - Scale the loss value, un-scale the gradients (in FP32!)
  - Check gradients at each iteration for overflow - adjust loss scale and skip update, if needed
AUTOMATIC MIXED PRECISION

Concepts

• Allows to implement the three-part methodology automatically:
  • The framework software can transform existing model code to run with mixed precision fully automatically
  • No new code required - should result in no new bugs

• Two components:
  • **Automated casting** operation-level logic to insert casts between FP32 and FP16, transparent to the user
  • **Automatic loss scaling**: wrapper class for the optimizer object that can scale the loss, keep track of the loss scale, and skip updates as necessary
ENABLING AMP
TensorFlow

- NVIDIA container 19.07+ and TF 1.14+:
  - We provide an explicit optimizer wrapper to perform loss scaling, which can also enable auto-casting for you:
    ```python
    opt = tf.train.experimental.enable_mixed_precision_graph_rewrite(opt)
    ```
  - See docs here:

- NVIDIA container 19.03+:
  - `TF_ENABLE_AUTO_MIXED_PRECISION=1` [automatic casting and automatic loss scaling]
  - `TF_ENABLE_AUTO_MIXED_PRECISION_GRAPH_REWRITE=1` [automatic casting only]
  - `TF_ENABLE_AUTO_MIXED_PRECISION_LOSS_SCAILING=1` [automatic loss scaling only]
• **Two steps:** initialization and wrapping backpropagation

```python
from apex import amp

model = ...
optimizer = torch.optim.SGD(model.parameters(), lr=1e-3)

model, optimizer = amp.initialize(model, optimizer, opt_level="O1")

# ...
for train_loop():
    loss = loss_fn(model(x), y)
    with amp.scale_loss(loss, optimizer) as scaled_loss:
        scaled_loss.backward()
    # Can manipulate the .grads if you’d like
    optimizer.step()
```
## OBTAINING MAXIMUM PERFORMANCE Cheatsheet

### Satisfy Shape Constraints
- Choose mini-batch to be a multiple of 8.
- Choose linear layer dimensions to be a multiple of 8.
- Choose convolution layer channel counts to be a multiple of 8.
- For classification problems, pad vocabulary to be a multiple of 8.
- For sequence problems, pad the sequence length to be a multiple of 8.

### Increase Arithmetic Intensity
- Concatenate weights and gate activations in recurrent cells.
- Concatenate activations across time in sequence models.
- Prefer dense math operations, e.g., vanilla convolutions have much higher arithmetic intensity than depth-wise separable convolutions.
- Prefer wider layers when possible accuracy-wise.

### Decrease Non-TC Work
- Try to minimize non-Tensor Core work - example: Accelerating a model with only 50% Tensor Core-accelerated operations (matrix multiplications and convolutions) by 5x only yields a total speedup of $1 / (0.5 + (0.5 / 5.)) = 1.67x$.
- Try to speed up non-Tensor Core ops with compiler tools such as XLA for TensorFlow and the PyTorch JIT.

MIXED PRECISION EXAMPLE

TIP #4 - THINK ABOUT DATA INGESTION
CPU BOTTLENECK OF DL TRAINING

Complexity of I/O pipeline

Alexnet
- 256x256 image
- 224x224 crop and mirror
- Training

ResNet 50
- 480p image
- Random resize
- Color augment
- 224x224 crop and mirror
- Training
CPU BOTTLENECK OF DL TRAINING

In practice

$2x$ GPUs doesn’t result in $2x$ perf improvement

Goal: $2x$

Reality: $< 2x$
Frameworks have their own I/O pipelines (often more than 1!)

Tradeoff between performance and flexibility

Lots of duplicated effort to optimize them all

Training process is not portable even if the model is (e.g. via ONNX)
SOLUTION: ONE LIBRARY

NVIDIA DATA LOADING LIBRARY (DALI)

• Centralize the effort
• Integrate into all frameworks
• Provide both flexibility and performance
NVIDIA DATA LOADING LIBRARY (DALI)
Fast Data Processing Library for Accelerating Deep Learning

Full input pipeline acceleration including data loading and augmentation

Drop-in integration with DL frameworks

Portable workflows through multiple input formats and configurable graphs

Flexible through configurable graphs and custom operators

Currently supports:
- ResNet50 (Image Classification), SSD (Object Detection)
- Input Formats - JPEG, LMDB, RecordIO, TFRecord, COCO, H.264, HVEC
- Python/C++ APIs to define, build & run an input pipeline

Over 1100 GitHub stars | Top 50 ML Projects (out of 22,000 in 2018)
DATA PIPELINE - BEFORE & AFTER

Framework Pre-processing – Without DALI

Before: All on CPU

Loader ➔ Decode ➔ Resize ➔ Augment ➔ Training

Framework Pre-processing – With DALI & nvJPEG

After: Mostly on GPU

Loader ➔ Decode ➔ Resize ➔ Augment ➔ Training

DALI pipeline runs on CPU and GPU
SET YOUR DATA FREE

DALI

LMDB (Caffe, Caffe2)
RecordIO (MXNet)
TFRecord (TensorFlow)
List of JPEGs (PyTorch, others)

Use any file format in any framework
USING DALI

Simple Python Interface to Implement Data Processing in a Few Steps

- Many examples & tutorials to help you get started
  - Image Classification (RN50) for Pytorch, MXNet, TensorFlow
  - How to read data in various frameworks
  - How to create custom operators
  - Pipeline for object detection
  - Multi-GPU training
  - Video pipeline...more to come

TIP #5 - USE NVLINK FOR MULTI-GPU TRAINING
WHAT IS NVLINK?
**MULTI GPU DATA PARALLEL DL TRAINING**

Mini-batch split between different GPUs
MULTI GPU DATA PARALLEL DL TRAINING

We also need to synchronise weights!
MULTI GPU DATA PARALLEL DL TRAINING

First load training data (different to each GPU)
MULTI GPU DATA PARALLEL DL TRAINING

Secondly, synchronize weights between the GPUs.
DL DATA PARALLELISM - PCIE BASED

Data loading over PCle
DL DATA PARALLELISM - PCIE BASED

Gradient averaging over PCle and QPI
Data loading and gradient averaging share communication resources: Congestion
DL DATA PARALLELISM - NVLINK
DL DATA PARALLELISM - NVLINK

Data loading over PCIe
DL DATA PARALLELISM - NVLINK

Gradient averaging over NVLink
DL DATA PARALLELISM - NVLINK

No sharing of communication resources: No congestion
HOROVOD

“Making distributed Deep Learning fast and easy to use”

- Leverages TensorFlow + MPI + NCCL to simplify development of synchronous multigpu/multinode TensorFlow
- Instead of Parameter Server architecture leverages MPI and NCCL based all reduce
- Owing to NCCL it leverages features such as:
  - NVLINK
  - RDMA
  - GPUDirectRDMA
  - Automatically detects communication topology
  - Can fall back to PCIe and TCP/IP communication
BENCHMARKING AFTER HOROVOD

2X Standard Distributed TensorFlow

OTHER TIPS

Honourable mentions

• Use TRTIS for inference
• Use tiered storage
• Prefer multi-threaded image libraries
• Use RAPIDS for ML workloads
• Read the Developer Blogs on the NV website
• Base your models on optimised examples in NGC containers