CNN INFERENCe WITH cuDNN
Chris Hebert, Sven Middelberg, March 21, 2019
Introduction to cuDNN

cuDNN Best Practices:
- Memory Management Done Right
- Choosing the Right Convolution Algorithm & Tensor Layout
- Tensor Cores: Low Precision Inference at Speed of Light

From Research to Production: It just works ... or not?!

Summary
cuDNN?

**cuDNN** provides a set of common network operations:

- Convolution
- Activation
- Tensor Ops: Add, multiply etc

Highly optimized for respective HW architectures.

**cuDNN** is the backend for most DL frameworks that target NVIDIA Hardware.
cudnnHandle_t ctx;
cudnnCreate(&ctx);

cudnnTensorDescriptor_t in_desc;
cudnnCreateTensorDescriptor(&in_desc);
cudnnSetTensorDescriptor(in_desc, CUDNN_TENSOR_NCHW, CUDNN_DATA_FLOAT, 1, 1920, 1080, 4);
// Same for out_desc

cudnnFilterDescriptor_t filter_desc;
cudnnCreateFilterDescriptor(&filter_desc);
cudnnSetFilter4dDescriptor(filter_desc, CUDNN_DATA_FLOAT, CUDNN_TENSOR_NCHW, 8, 4, 3, 3);

cudnnConvolutionDescriptor_t conv_desc;
cudnnCreateConvolutionDescriptor(&conv_desc);
cudnnSetConvolution2dDescriptor(conv_desc, 1, 1, 2, 2, 1, 1, CUDNN_CONVOLUTION, CUDNN_DATA_FLOAT);

size_t workspace_size; void* workspace;
cudnnGetConvolutionForwardWorkspaceSize(ctx, in_desc, filter_desc, conv_desc, out_desc, conv_alg, &workspace_size);
cudaMalloc(&workspace, workspace_size);

// Allocate and initialize device data ...

float alpha = 1.0f; float beta = 0.0f;
cudnnStatus_t status = cudnnConvolutionForward(ctx, &alpha, in_desc, in_dev, filter_desc, weights_dev, conv_desc, conv_alg, workspace, workspace_size, &beta, out_desc, out_dev);
cuDNN has no native support for fully-connected layers

Forward pass of fully-connected layers is basically a matrix vector multiply

\[
\begin{pmatrix}
 v_0 \\
 v_1 \\
 v_2
\end{pmatrix}
\begin{pmatrix}
 w_0 & w_1 & w_2
\end{pmatrix}
\begin{pmatrix}
 a_0 \\
 a_1 \\
 a_2
\end{pmatrix} =
\begin{pmatrix}
 b_0 \\
 b_1
\end{pmatrix}
\]

➢ cuBLAS
cuDNN PROFILING

Profiling Tools

- NVIDIA Nsight Systems / NVVP
- nvprof
- NVIDIA Nsight Compute
  Custom CUDA kernel profiling

GPU timing using CUDA events:

```c
cudaEvent_t start, stop;
cudaEventCreate(&start); cudaEventCreate(&stop);

cudaEventRecord(start);
// Do something on GPU
cudaEventRecord(stop);

cudaEventSynchronize(stop);

float ms;
cudaEventElapsedTime(&ms, start, stop);
```
TensorRT & cuDNN
## TensorRT & cuDNN

<table>
<thead>
<tr>
<th>TensorRT</th>
<th>cuDNN</th>
</tr>
</thead>
<tbody>
<tr>
<td>High: Uses cuDNN internally</td>
<td>Low: Basic DL primitives</td>
</tr>
<tr>
<td>Low: Only custom layers must be implemented manually</td>
<td>High: Network must be assembled manually</td>
</tr>
<tr>
<td>Level of abstraction</td>
<td>Programming effort</td>
</tr>
</tbody>
</table>
## TensorRT & cuDNN

<table>
<thead>
<tr>
<th>TensorRT</th>
<th>cuDNN</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>High:</strong> Uses cuDNN internally</td>
<td><strong>Low:</strong> Basic DL primitives</td>
</tr>
<tr>
<td><strong>Low:</strong> Only custom layers must be implemented manually</td>
<td><strong>High:</strong> Network must be assembled manually</td>
</tr>
<tr>
<td><strong>High:</strong> Allows to import models from many training frameworks</td>
<td><strong>None:</strong> Models must be imported manually</td>
</tr>
</tbody>
</table>

- **Level of abstraction**
- **Programming effort**
- **Compatibility support for other APIs**
## TensorRT & cuDNN

<table>
<thead>
<tr>
<th>TensorRT</th>
<th>cuDNN</th>
</tr>
</thead>
<tbody>
<tr>
<td>High: Uses cuDNN internally</td>
<td>Low: Basic DL primitives</td>
</tr>
<tr>
<td>Low: Only custom layers must be implemented manually</td>
<td>High: Network must be assembled manually</td>
</tr>
<tr>
<td>High: Allows to import models from many training frameworks</td>
<td>None: Models must be imported manually</td>
</tr>
<tr>
<td>Low: Fully-automatic inference optimization</td>
<td>High: Needs extensive profiling and optimization</td>
</tr>
</tbody>
</table>

- **Level of abstraction**
- **Programming effort**
- **Compatibility support for other APIs**
- **Optimization effort**
# TensorRT & cuDNN

<table>
<thead>
<tr>
<th>TensorRT</th>
<th>cuDNN</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>High:</strong> Uses cuDNN internally</td>
<td><strong>Low:</strong> Basic DL primitives</td>
</tr>
<tr>
<td><strong>Low:</strong> Only custom layers must be implemented manually</td>
<td><strong>High:</strong> Network must be assembled manually</td>
</tr>
<tr>
<td><strong>High:</strong> Allows to import models from many training frameworks</td>
<td><strong>None:</strong> Models must be imported manually</td>
</tr>
<tr>
<td><strong>Low:</strong> Fully-automatic inference optimization</td>
<td><strong>High:</strong> Needs extensive profiling and optimization</td>
</tr>
<tr>
<td><strong>Good</strong></td>
<td><strong>Potentially better (depending on effort)</strong></td>
</tr>
</tbody>
</table>

- **Level of abstraction**
- **Programming effort**
- **Compatibility support for other APIs**
- **Optimization effort**
- **Performance**
# TensorRT & cuDNN

<table>
<thead>
<tr>
<th><strong>TensorRT</strong></th>
<th><strong>cuDNN</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>High:</strong> Uses cuDNN internally</td>
<td><strong>Low:</strong> Basic DL primitives</td>
</tr>
<tr>
<td><strong>Low:</strong> Only custom layers must be implemented manually</td>
<td><strong>High:</strong> Network must be assembled manually</td>
</tr>
<tr>
<td><strong>High:</strong> Allows to import models from many training frameworks</td>
<td><strong>None:</strong> Models must be imported manually</td>
</tr>
<tr>
<td><strong>Low:</strong> Fully-automatic inference optimization</td>
<td><strong>High:</strong> Needs extensive profiling and optimization</td>
</tr>
<tr>
<td><strong>Good</strong></td>
<td><strong>Potentially better (depending on effort)</strong></td>
</tr>
<tr>
<td><strong>Low:</strong> Re-optimization needed for any change in architecture or input dimensions</td>
<td><strong>High:</strong> Usually requires very few optimizations after reasonable changes in input / architecture</td>
</tr>
</tbody>
</table>

**Level of abstraction**

**Programming effort**

**Compatibility support for other APIs**

**Optimization effort**

**Performance**

**Mutability**
Introduction to cuDNN

cuDNN Best Practices:
- Memory Management Done Right
- Choosing the Right Convolution Algorithm & Tensor Layout
- Tensor Cores: Low Precision Inference at Speed of Light

From Research to Production: It just works ... or not?!

Summary
AUTOENCODER IN CAFFE
Deep Image Matting, Chris Hebert, GTC‘18

160ms
AUTOENCODER IN CAFFE
Deep Image Matting, Chris Hebert, GTC‘18

160ms
CUDNN DEVICE MEMORY MANAGEMENT
Minimize Footprint - Maximize Performance

Don’ts

✗ Be wasteful with allocations
✗ Interleaving allocations with kernel execution
✗ Memcpy layer weights on-the-fly, one H2D-copy per layer per forward pass
✗ Copy synchronously from pageable host memory

Dos

 ✓ Maximize reuse
 ✓ Allocate once at startup, use every forward pass
 ✓ Initialize layer weights once, copy only the input and output tensors per forward pass
 ✓ Copy asynchronously from page-locked host memory, maximize overlap and resource utilization
INITIALIZING WEIGHTS AT STARTUP
Deep Image Matting, Chris Hebert, GTC‘18

Overall weights data transferred for each inference is \(~89\) MB ⇒ 8.7 ms @10Gb/s

Allocate and initialize once at startup, reuse each inference!
Combined size of all output tensors is **141.8 MB**

Only two tensors used at a time: input & output tensor

- Allocate two times the maximum tensor size: **50 MB**
MINIMIZING MEMORY FOOTPRINT

“Ping-Pong” Tensor Memory

Conv1_1
Conv1_2
Conv1_3
Pool1
Conv2_1

Memory Pool
2x Largest Tensor

A 25mb
B 25mb

Doesn't work for cached tensors, e.g., skip links!
MINIMIZING MEMORY FOOTPRINT

Workspace Memory

Size of a convolution workspace varies, depending on multiple parameters:

• input and output tensor dimensions
• Precisions
• Convolution algorithm
• ...

But: workspace can be shared among layers

Allocate maximum workspace size!
OPTIMIZING RESOURCE UTILIZATION

Maximum Inference Throughput by Asynchronous Copies

Use page-locked host memory for H2D and D2H memcpys:

\[ \text{cudaMallocHost}(\ldots) \text{ instead of } \text{malloc}(\ldots) \]

Use asynchronous memcpys:

\[ \text{cudaMemcpyAsync}(\ldots, \text{stream}) \text{ instead of } \text{cudaMemcpy}(\ldots) \]

Use multiple CUDA streams to overlap memcpys and kernel executions:

- H2D copy of input tensor
- Forward pass kernels
- D2H copy of output tensor

- 0 1 2 3 4 5 6 7
- 0 1 2 3 4 5 6 7
- 0 1 2 3 4 5 6 7
AUTOENCODER IN CAFFE
Deep Image Matting, Chris Hebert, GTC’18

160ms
AUTOENCODER IN cuDNN
Deep Image Matting, Chris Hebert, GTC‘18

Start-up time (one time overhead) 41ms
17ms
AGENDA

Introduction to cuDNN

cuDNN Best Practices:
- Memory Management Done Right
- Choosing the Right Convolution Algorithm & Tensor Layout
- Tensor Cores: Low Precision Inference at Speed of Light

From Research to Production: It just works … or not?!

Summary
### CONVOLUTION ALGORITHMS

128x128x128x128 convolution, FP32, NCHW, Quadro GV100

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>3 x 3</th>
<th>11 x 11</th>
</tr>
</thead>
<tbody>
<tr>
<td>Performance</td>
<td></td>
<td>Performance</td>
</tr>
<tr>
<td>Workspace</td>
<td></td>
<td>Workspace</td>
</tr>
</tbody>
</table>

- CUDNN_CONVOLUTION_FWD_ALGO_GEMM
- CUDNN_CONVOLUTION_FWD_ALGO_IMPLICIT_GEMM
- CUDNN_CONVOLUTION_FWD_ALGO_IMPLICIT_PRECOMP_GEMM
- CUDNN_CONVOLUTION_FWD_ALGO_FFT
- CUDNN_CONVOLUTION_FWD_ALGO_FFT_TILING
- CUDNN_CONVOLUTION_FWD_ALGO_WINOGRAD
- CUDNN_CONVOLUTION_FWD_ALGO_WINOGRAD_NONFUSED
## Convolution Algorithms

128x128x128x128 convolution, FP32, NCHW, Quadro GV100

<table>
<thead>
<tr>
<th>Algorithm Name</th>
<th>Performance</th>
<th>Workspace</th>
<th>Performance</th>
<th>Workspace</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUDNN_CONVOLUTION_FWD_ALGO_GEMM</td>
<td>0.76 ms</td>
<td>72 MB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CUDNN_CONVOLUTION_FWD_ALGO_IMPLICIT_GEMM</td>
<td>0.62 ms</td>
<td>None</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CUDNN_CONVOLUTION_FWD_ALGO_IMPLICIT_PRECOMP_GEMM</td>
<td>0.47 ms</td>
<td>0.01 MB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CUDNN_CONVOLUTION_FWD_ALGO_FFT</td>
<td>45.3 ms</td>
<td>8322 MB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CUDNN_CONVOLUTION_FWD_ALGO_FFT_TILING</td>
<td>3.69 ms</td>
<td>70 MB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CUDNN_CONVOLUTION_FWD_ALGO_WINOGRAD</td>
<td>0.26 ms</td>
<td>1.56 MB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CUDNN_CONVOLUTION_FWD_ALGO_WINOGRAD_NONFUSED</td>
<td>2.73 ms</td>
<td>578 MB</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## CONVOLUTION ALGORITHMS

128x128x128x128 convolution, FP32, NCHW, Quadro GV100

<table>
<thead>
<tr>
<th>CUDNN_CONVOLUTION_FWD_ALGO_ ...</th>
<th>3 x 3</th>
<th>11 x 11</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Performance</td>
<td>Workspace</td>
</tr>
<tr>
<td>CUDNN_CONVOLUTION_FWD_ALGO_GEMM</td>
<td>0.76 ms</td>
<td>72 MB</td>
</tr>
<tr>
<td>CUDNN_CONVOLUTION_FWD_ALGO_IMPLICIT_GEMM</td>
<td>0.62 ms</td>
<td>None</td>
</tr>
<tr>
<td>CUDNN_CONVOLUTION_FWD_ALGO_IMPLICIT_PRECOMP_GEMM</td>
<td>0.47 ms</td>
<td>0.01 MB</td>
</tr>
<tr>
<td>CUDNN_CONVOLUTION_FWD_ALGO_FFT</td>
<td>45.3 ms</td>
<td>8322 MB</td>
</tr>
<tr>
<td>CUDNN_CONVOLUTION_FWD_ALGO_FFT_TILING</td>
<td>3.69 ms</td>
<td>70 MB</td>
</tr>
<tr>
<td>CUDNN_CONVOLUTION_FWD_ALGO_WINOGRAD</td>
<td>0.26 ms</td>
<td>1.56 MB</td>
</tr>
<tr>
<td>CUDNN_CONVOLUTION_FWD_ALGO_WINOGRAD_NONFUSED</td>
<td>2.73 ms</td>
<td>578 MB</td>
</tr>
</tbody>
</table>
CONVOLUTION ALGORITHMS

Choice depends on memory and performance requirements

Some algorithms not supported for certain convolution configurations

Choosing the most suitable and supported algorithm, layer by layer, is a tedious job

- `cudnnGetConvolutionForwardAlgorithm_v7(...)`
  - Based on heuristics: List of algorithms sorted by expected runtime

- `cudnnFindConvolutionForwardAlgorithm(...)`
  - More accurate results based on exhaustive experiments
# Tensor Layout

**NCHW vs NHWC**

<table>
<thead>
<tr>
<th>Input Tensor Size</th>
<th>Output Tensor Size</th>
<th>Filter Size</th>
<th>NCHW</th>
<th>NHWC</th>
</tr>
</thead>
<tbody>
<tr>
<td>32 x 32 x 64</td>
<td>16 x 16 x 128</td>
<td>3 x 3</td>
<td>0.05 ms</td>
<td>0.06 ms</td>
</tr>
<tr>
<td>128 x 128 x 128</td>
<td>128 x 128 x 128</td>
<td>3 x 3</td>
<td>0.26 ms</td>
<td>0.50 ms</td>
</tr>
<tr>
<td>512 x 512 x 32</td>
<td>256 x 256 x 64</td>
<td>5 x 5</td>
<td>0.56 ms</td>
<td>1.06 ms</td>
</tr>
<tr>
<td>1920 x 1080 x 3</td>
<td>1920 x 1080 x 32</td>
<td>5 x 5</td>
<td>0.97 ms</td>
<td>1.36 ms</td>
</tr>
<tr>
<td>16 x 16 x 128</td>
<td>8 x 8 x 256</td>
<td>7 x 7</td>
<td>0.40 ms</td>
<td>0.41 ms</td>
</tr>
<tr>
<td>1920 x 1080 x 4</td>
<td>960 x 540 x 32</td>
<td>9 x 9</td>
<td>1.22 ms</td>
<td>1.34 ms</td>
</tr>
<tr>
<td>128 x 128 x 128</td>
<td>128 x 128 x 128</td>
<td>11 x 11</td>
<td>5.13 ms</td>
<td>5.72 ms</td>
</tr>
</tbody>
</table>

Convolution algorithm selected using `cudnnFindConvolutionForwardAlgorithm(...)`. 
TENSOR LAYOUT

Usually NCHW is faster than NHWC for tensor data

Might be reasonable to pre-convert NHWC input tensor to NCHW and back after the inference to achieve optimal throughput

⇒ Profile!

One exception ...
AGENDA

Introduction to cuDNN

cuDNN Best Practices:
- Memory Management Done Right
- Choosing the Right Convolution Algorithm & Tensor Layout
- Tensor Cores: Low Precision Inference at Speed of Light

From Research to Production: It just works ... or not?!

Summary
LOW PRECISION INFERENCE

In most cases FP16 / half provides more than adequate precision for image processing.

Volta and Turing have hardware for FAST fp16 - TRUE_HALF_CONFIG

On Pascal and below, store in fp16 but process in fp32 - PSEUDO_HALF_CONFIG

Given an FP32 model, simply converting the weights to FP16 often retains decent quality.

For best results ⇒ Retrain with FP16 precision.
LOW PRECISION INFERENCE
Deep Image Matting, Chris Hebert, GTC‘18

![Speedup vs Input Size](chart.png)
LOW PRECISION INFERENCE

Deep Image Matting, Chris Hebert, GTC‘18

Memory Demands

- **Input Size**: 320x320, 640x640, 960x960, 1280x1280
- **Precision**: FP32, FP16
Tensor Cores perform FP16 matrix multiply accumulate (HMMA)

Turing also supports INT8 and INT4

Only two algorithms supported:

- `CUDNN_CONVOLUTION_FWD_ALGO_WINOGRAD_NONFUSED`
- `CUDNN_CONVOLUTION_FWD_ALGO_IMPLICIT_PRECOMP_GEMM`

Number of Input and output channels must be multiple of eight!

Convolution math type must be set to `CUDNN_TENSOR_OP_MATH`

And it will be much MUCH faster!
LOW PRECISION INFEERENCE

Deep Image Matting, Chris Hebert, GTC‘18

![Bar chart showing speedup for different input sizes and precision levels. The x-axis represents input size (320x320, 640x640, 960x960, 1280x1280), and the y-axis represents speedup. The chart compares three precision levels: FP32, FP16, and FP16 + TC.](image-url)
LOW PRECISION INFERENCE
Deep Image Matting, Chris Hebert, GTC‘18

Input Size

Memory Demands

FP32  FP16  FP16 + TC

320x320  640x640  960x960  1280x1280
# TENSOR CORES ON VOLTA AND TURING

## NCHW vs NHWC

<table>
<thead>
<tr>
<th>Input Tensor Size</th>
<th>Output Tensor Size</th>
<th>Filter Size</th>
<th>NCHW</th>
<th>NHWC</th>
</tr>
</thead>
<tbody>
<tr>
<td>32 x 32 x 64</td>
<td>16 x 16 x 128</td>
<td>3 x 3</td>
<td>0.05 ms</td>
<td>0.04 ms</td>
</tr>
<tr>
<td>128 x 128 x 128</td>
<td>128 x 128 x 128</td>
<td>3 x 3</td>
<td>0.11 ms</td>
<td>0.08 ms</td>
</tr>
<tr>
<td>512 x 512 x 32</td>
<td>256 x 256 x 64</td>
<td>5 x 5</td>
<td>0.25 ms</td>
<td>0.15 ms</td>
</tr>
<tr>
<td>1920 x 1080 x 8</td>
<td>1920 x 1080 x 32</td>
<td>5 x 5</td>
<td>3.00 ms</td>
<td>2.31 ms</td>
</tr>
<tr>
<td>16 x 16 x 128</td>
<td>8 x 8 x 256</td>
<td>7 x 7</td>
<td>0.26 ms</td>
<td>0.23 ms</td>
</tr>
<tr>
<td>128 x 128 x 128</td>
<td>128 x 128 x 128</td>
<td>7 x 7</td>
<td>0.37 ms</td>
<td>0.34 ms</td>
</tr>
<tr>
<td>800 x 800 x 8</td>
<td>400 x 400 x 8</td>
<td>9 x 9</td>
<td>1.20 ms</td>
<td>2.53 ms</td>
</tr>
</tbody>
</table>

Convolution algorithm selected using `cudnnFindConvolutionForwardAlgorithm(…)`
CHANNEL PADDING

Some tensors might not have a channel count that is a multiple of eight, e.g., three-channel RGB input tensors

- Cannot use Tensor Core acceleration

512x512x3 → 512x512x32 convolution, 3x3 kernel, FP16, NHWC, GV100: 0.84 ms

512x512x8 → 512x512x32 convolution, 3x3 kernel, FP16, NHWC, GV100: 0.18 ms

- Pad input tensor, zero-pad filter weights
Tensor Core convolution performance varies with channel count

Both of these 3x3 convolutions uses input and output tensors that have the same size:

\[
\begin{align*}
2048 \times 2048 \times 8 & \rightarrow 2048 \times 2048 \times 8 \quad \Rightarrow \quad 2.01 \text{ ms} \quad 24 \text{ MB} \\
1024 \times 1024 \times 32 & \rightarrow 1024 \times 1024 \times 32 \quad \Rightarrow \quad 0.6 \text{ ms} \quad 6 \text{ MB}
\end{align*}
\]

- Fold 2x2xN slices into 1x1x4N slices

- Increases receptive field

- Requires re-training
Introduction to cuDNN

cuDNN Best Practices:

- Memory Management Done Right
- Choosing the Right Convolution Algorithm & Tensor Layout
- Tensor Cores: Low Precision Inference at Speed of Light

From Research to Production: It just works … or not?!

Summary
Research To Production

What do we need?

(e.g.) TensorFlow, PyTorch

Network Definition

Network Parameters
Research To Production

What do we need?

(e.g.) TensorFlow, PyTorch

Network Definition

Network Parameters

A few ways to do this.
Research To Production

C++ Parse The Protocol Buffers

- Most checkpoint/model file formats based on Protocol Buffers from Google
  - Check em out, they’re awesome.
- Message format defined in the .proto file
- Compiled with the Protocol Buffer Compiler
- Manipulate the contents with the Protocol Buffer API
- Good tutorials for this at
  - https://developers.google.com/protocol-buffers/docs/cpptutorial
**Research To Production**

Write the params and arch straight from Python

**PyTorch example (simplified)**

Load the model in Python, open output file for writing

```
......
head_0_spade_0_mlp_shared_0, Weights, 176198144, 2342400, 128, 183, 5, 5
head_0_spade_0_mlp_shared_0, biases, 176197632, 512, 1, 1, 1, 128
......
```

<table>
<thead>
<tr>
<th>Tensor Name</th>
<th>Offset,Size</th>
<th>Shape</th>
</tr>
</thead>
<tbody>
<tr>
<td>head_0_spade_0_mlp_shared_0, Weights</td>
<td>176198144, 2342400</td>
<td>128, 183, 5, 5</td>
</tr>
<tr>
<td>head_0_spade_0_mlp_shared_0, biases</td>
<td>176197632, 512</td>
<td>1, 1, 1, 128</td>
</tr>
</tbody>
</table>
Research To Production
Write the model architecture straight from Python

PyTorch example (simplified)

Load the model in Python, open output file for writing

```python
input_file_path = <Path to Pytorch checkpoint>
ckpnt = torch.load(input_file_path,map_location="cpu")
out_path = "netG_params.txt".format(item_key)
    with open(out_path,"w") as f:
```
Research To Production
Write the model architecture straight from Python

PyTorch example (simplified)
Iterate the model, find the weights and biases

```python
input_file_path = <Path to Pytorch checkpoint>
ckpnt = torch.load(input_file_path,map_location="cpu")
out_path = "netG_params.txt".format(item_key)
with open(out_path,"w") as f:
    for model_key in ckpnt :
        if model_key.find("weight") > 0 or model_key.find("bias") > 0:
            cur_var = Variable(ckpnt[model_key])
            var_size = cur_var.size()
            size_len = len(var_size)
```
Research To Production
Write the model architecture straight from Python

PyTorch example (simplified)
Replace ‘.’ with ‘_’ (personal preference)

tensor_name = model_key.replace(".","_")
tensor_type = ""
if model_key.find("weight") > 0:
    tensor_type = "Weights"
    tensor_name = tensor_name.replace("_weight","")
if model_key.find("bias") > 0:
    tensor_type = "biases"
    tensor_name = tensor_name.replace("_bias","")
tensor_dims = ""
tensor_total_size = 1
PyTorch example (simplified)

Record the tensor shape in a consistent manner

```python
if size_len < 4:
    size_len_delta = 4-size_len
    for s in range(size_len_delta):
        tensor_dims += ",1"

    for s in range(size_len):
        tensor_dims += ",{},".format(var_size[s])
        tensor_total_size *= var_size[s]

tensor_total_file_size = tensor_total_size * 4

tensor_size_data = ",{},{}".format(tensor_offset, tensor_total_file_size)
```
Research To Production
Write the model architecture straight from Python

PyTorch example (simplified)

Write the name, offset, size and shape to the text file.

tensor_total_file_size = tensor_total_size * 4
tensor_size_data = "{},{},{}".format(tensor_offset,tensor_total_file_size)
tensor_offset += tensor_total_file_size
tensor_name += "{},{}".format(tensor_type)
f.write(tensor_name)
f.write(tensor_size_data)
f.write(tensor_dims)
f.write("\n")
**Research To Production**

Write the params and arch straight from Python

---

**PyTorch example (simplified)**

Load the model in Python, open output file for writing

```
......
head_0_spade_0_mlp_shared_0,Weights, 176198144,2342400, 128,183,5,5
head_0_spade_0_mlp_shared_0,biases, 176197632,512, 1,1,1,128
......
```

<table>
<thead>
<tr>
<th>Tensor Name</th>
<th>Offset,Size</th>
<th>Shape</th>
</tr>
</thead>
<tbody>
<tr>
<td>head_0_spade_0_mlp_shared_0,Weights</td>
<td>176198144,2342400</td>
<td>128,183,5,5</td>
</tr>
<tr>
<td>head_0_spade_0_mlp_shared_0,biases</td>
<td>176197632,512</td>
<td>1,1,1,128</td>
</tr>
</tbody>
</table>
Research To Production
Write the params and arch straight from Python

PyTorch example (simplified)

Load the model in Python, open output file for writing

......
head_0_spade_0_mlp_shared_0,Weights, 176198144,2342400, 128,183,5,5
head_0_spade_0_mlp_shared_0,biases, 176197632,512, 1,1,1,128
......

Tensor Name  Offset,Size  Shape
PyTorch example (simplified)

Similar loop as before but extract the weights from the .data member and write to a single file

```python
(iterate model as before)

data = ckpt[model_key]
cur_var = Variable(data)
var_size = cur_var.size()
np_data = data.cpu().numpy()
f.write(np_data.tobytes())
```
Scientists express their models in an algebraically correct manner.

- That’s their job.
- But algebraically correct does not necessarily mean performant.

Engineers need to identify when an algorithm can be restructured for performance.

- That’s our job.
Research To Production

Scientist vs Engineer

Example 1. $Wx + b$ when you ONLY want the bias.
Example 1. \( Wx + b \) when you ONLY want the bias.
Example 1. \( Wx+b \) when you ONLY want the bias.

In this particular case:

- Write custom kernel to write the bias values.
- And if possible fuse with previous and/or next step.
Example 2. Downsampling called many times on the same data.
Example 2. Downsampling called many times on the same data.

Layer Block (e.g. Resnet)
- Sub Layer 0 - (e.g. Normalization)
- Sub Layer 1 -
- Sub Layer S - (ResNet Shortcut)

Sub Layer
- Downsampling original Input
- Do Layer Specific Stuff

This is the same operation on the same data 3 times
Research To Production
Scientist vs Engineer

Example 2. Re order the operations....

Layer Block (e.g. Resnet)

Downsample original Input

- Sub Layer 0 - (e.g. Normalization)
- Sub Layer 1 -
- Sub Layer S - (ResNet Shortcut)

Sub Layer

Do Layer Specific Stuff

This is the same operation on the same data 3 times
Research To Production
Porting to TensorRT

TensorFlow PyTorch Caffe etc. → 3rd Party Utils → ONNX

Trt Python tools → UFF

Weights and Architecture → A bit of elbow grease → C++ API

TensorRT
Research To Production
UFF, ONNX or API .... Which to use....

- Most common architectures will import directly from TensorFlow/PyTorch etc
- Most common operations are already supported in TensorRT
- Convolution/Cross Correlation
- Activation
  - Sigmoid, Relu, Clipped Relu, TanH, ELU
- Batch Norm
  - Spatial, Spatial_persistent, Per Activation
- Pooling
  - Max, Average
Research To Production
UFF, ONNX or API .... Which to use....

• Sometimes it’s not that easy

• Sometimes some graph surgery is required.
  • Edit the graph to strip out e.g. pre/post processing at either end of the graph

• TensorRT provides a plug-in interface for custom layers
  • Name custom layers as per the incoming model (e.g. LeakyRelu)
  • From TrT 5.1 : The IPlugInV2 interface supports optimization.

• There is a simpler option
Research To Production
Porting to TensorRT Using TfTrt

- Converts TensorFlow graph into 1 or more TensorRT ‘blocks’
- Adds these blocks back onto TensorFlow graph
- Inference of these blocks performed with TensorRT
- The rest use TensorFlow

Workflow:
- Load TensorFlow graph
- Prepare for inference (freeze layers, convert variables to constants etc)
- Call trt.create_inference_graph(input_graph_def, outputs, max_batch_size, max_workspace_size, precision_mode)
Research To Production
Porting to TensorRT Using TfTrt

TensorFlow
- TensorRT friendly
- Custom Stuff
- TensorRT friendly
- Custom Stuff
- TensorRT friendly

TfTrt

TensorRT block
- Custom Stuff
- TensorRT block
- Custom Stuff
- TensorRT block

Still TensorFlow

TensorRT ‘blocks’ optimized and added back onto the TensorFlow Graph
Research To Production
Porting ‘Funky’ networks to TensorRT

Important takeaways from this

You don’t need generate a single monolithic graph with TensorRT.

Generate graph snippets from TensorRT interleaved with custom CUDA.

You can do this with the TensorRT API.

Run them in whatever sequence you need at run time.

Allows you to create inference solutions with dynamic runtime behavior.

Keep all data on the GPU whenever possible.
Research To Production

When it doesn’t ....... Just work.

Here’s a debugging tip...

Original Pytorch/Tf
Working 😊

TensorRT or cuDNN
Not working 😞
Research To Production

When it doesn’t …… Just work.

Here’s a debugging tip...

Original Pytorch/Tf Working 😊

TensorRT or cuDNN Not working 😞

Dump Layer Outputs to Disk
Research To Production
When it doesn’t …… Just work.

Here’s a debugging tip...

Original Pytorch/Tf Working 😊

TensorRT or cuDNN Not working 😞

Compare tensors with Python script.

Narrow down the problem to here.
AGENDA

Introduction to cuDNN

cuDNN Best Practices:
- Memory Management Done Right
- Choosing the Right Convolution Algorithm & Tensor Layout
- Tensor Cores: Low Precision Inference at Speed of Light

From Research to Production: It just works … or not?!

Summary
SUMMARY

Common DL frameworks often far from optimized for inference on GPUs

➢ Use cuDNN (or TensorRT) if you care about performance & memory!

Memory Management matters!

Lower your precision if possible!

Use hardware-specific optimizations, e.g. Tensor Cores on Volta & Turing!

You can never profile too much!

Download, documentation, discussion board, ...

www.developer.nvidia.com/cudnn