NVIDIA DEEP LEARNING PROFILER
GOAL

1. Why Profiling
2. What is DLProf
3. Use-case example
Deep Learning Profiling
DL PROFILING NEEDS
Profiling needs for different personas

Researchers
Fast development of best performant models for research, challenge and domains

Data Scientists & Applied Researchers
Reduce Training time, focus on data, develop and apply the best models for the applications

Software developers
Sysadmins & DevOps
Optimized utilization and uptime, monitor GPU workloads, leverage hardware
FINDING DL PERFORMANCE OPPORTUNITIES

Models can be bound by the data pipeline, compute or memory

- GPU utilization as it relates to model code
  - Time being spent on ops in every iteration
  - Time spent on GPU/CPU
  - Data types used for operations
- Bottlenecks could be attributed to
  - Input data pipeline: data loading, preprocessing etc
  - Compute (math) limited operations
  - Memory limited operations
  - Other aspects such as overall system tuning
- Categories of operations in DNNs based on bottleneck
  - Element wise: ReLU, memory bound
  - Reduction: Batch norm, memory bound
  - Dot product: Convolution, math bound

Compute bound

- conv → conv → conv → FC

Memory bound

- bias → norm → relu → dropout

Compute heavy ops see speed-ups from GPUs
NVIDIA PROFILING TOOLS
Nsight tools

Nsight Systems
Comprehensive workload-level performance

Nsight Compute
Detailed CUDA kernel performance

Nsight Graphics
Detailed frame/render performance

Start here

Dive into top CUDA kernels by using metrics/counter collection

Dive into graphics frames

Re-check overall performance
Why DLProf

- How can I profile a DL model?
  - I want to correlate the profiling data to the layers of my model and iterations
- I need aggregated information to understand the performance of my DL training process
  - Am I using GPUs properly?
  - What are the Ops/kernels that take the most GPU time?
  - How can I resolve the bottlenecks?
- I want to visualize the results intuitively
WHY DLPROF

Created for DL profiling
  • Correlation to model layer and iterations

FW agnostic
  • Supports multiple FWs

Ease of use
  • Pre-installed in NGC FW containers or python wheel
  • Prepend training script with `dlprof`

Offers details for power users
  • Provides actionable high-level information for DL workloads
  • Presents detailed information for power users as well
NVIDIA PROFILING STACK
The layers that make the cake

- **Nsight Systems** and **Nsight Compute** have been built using CUDA Profiling Tools Interface (CUPTI)
- **NVTX** NVIDIA Tools Extension Library is a way to annotate source code with markers
- NVTX markers are used to annotate and focus on sections of code important to the user
- TensorFlow optimized by NVIDIA (aka nvidia-tensorflow) contain support for NVTX markers.
- NVTX plugins are python bindings for users to add markers easily
- **DLProf** calls Nsight systems to collect profile data and correlate with the DL model

**NVIDIA COMPUTING PLATFORM** *(NGC Optimized Framework Containers)*

<table>
<thead>
<tr>
<th>Deep Learning Profiler (DLProf)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nsight Systems</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>NVTX for Tensorflow</th>
<th>NVTX Plugins</th>
<th>NVTX for PyTorch</th>
</tr>
</thead>
<tbody>
<tr>
<td>(NGC Optimized Framework Containers)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**DLProf Viewer**

| NVIDIA COMPUTING PLATFORM |
NVIDIA Deep Learning Profiler
DEEP LEARNING PROFILER

Deep Learning Profiler (DLProf) is a tool for profiling deep learning models

DLProf (CLI & Viewer)
• Helps data scientists understand and improve performance of their DL models by analyzing text reports or visualizing the data

DLProf CLI
• Uses Nsight Systems profiler under the hood
• Aggregates and correlates CPU and GPU profiling data from a training run to DL model
• Provides accurate Tensor Core usage detection for operations and kernels
• Identifies performance issues and provides recommendations via Expert Systems

DLProf Viewer
• Uses the results from DLProf CLI and provides visualization of the data
• Currently exists as a TensorBoard plugin
DLPROF OVERVIEW

“Are my GPUs being utilized?” “Am I using Tensor Cores?” “How can I improve performance?”

FW Support: TF1, TF2, PyT, and TRT
Lib Support: DALI, NCCL

Visualize Analysis and Recommendations
PREREQUISITES & INSTALLATION

NVIDIA optimized FW containers from NGC or pip wheel installation

NVIDIA Optimized FW containers from NGC

DLProf is distributed as a part of the PyTorch and TensorFlow (1.x and 2) containers on NGC


Python Wheel Installation

DLProf is also available as a pip wheel for PyTorch or TF 1.x

- Documentation link
- PyTorch example:
  - Install Nvidia Py Index:
    $ pip install nvidia-pyindex
  - Install DLProf with NVIDIA Pytorch, PyProf, and dependencies:
    $ pip install nvidia-dlprof[pytorch]
  - Install DLProf Viewer Plugin for TensorBoard:
    $ pip install nvidia-tensorboard-plugin-dlprof
GETTING STARTED
Running DLProf with TensorFlow, TRT, and PyTorch

1. TensorFlow and TRT require **no additional code modification**
2. Profile using DLProf CLI - prepend with `dlprof`
3. Visualize results with DLProf Viewer

1. Add few lines of code to your training script to enable `nvidia_dlprof_pytorch_nvtx` module
2. Profile using DLProf CLI - prepend with `dlprof`
3. Visualize with DLProf Viewer
GETTING STARTED
Running DLProf with TensorFlow, TRT, and PyTorch

1. Add few lines of code to your training script to enable `nvidia_dlprof_pytorch_nvtx` module
2. Profile using DLProf CLI - prepend with `dlprof`
3. Visualize with DLProf Viewer

Add few lines of code ➔ Profile ➔ Visualize in viewer
PROFILE OUTPUTS
Profile once, analyze output with multiple tools

Text reports
DLProf CLI can generate the following reports in CSV and JSON formats (*automation/scripting*)
- Summary
- Detail
- Iteration
- Kernel
- Tensor
- Node Op
- Group Node

Visual reports
DLProf CLI generates DLProf sqlite file that can be consumed by the DLProf Viewer

Nsight Systems reports
DLProf CLI generates a `nsys_profile.qdrep` and `nsys_profile.sqlite` that can be consumed and visualized by Nsight Systems UI
Deep Learning Profiler Viewer
• **GPU Utilization Chart:**
  - Shows the percentage of the wall clock time that the GPU is active. For multi-gpu, it is an average utilization across all GPUs

• **Op GPU Time Chart:**
  - Splits all operations into 3 categories: Operations that used tensor cores, operations that were eligible to use tensor cores but didn't, and operations that were ineligible to use tensor cores

• **Kernel GPU Time Chart:**
  - Breaks down all kernel time into 3 categories: Kernels that used tensor cores, memory kernels, and all other kernels

• **Tensor Core Kernel Efficiency Chart:**
  - Gives a single number that measures what percentage of GPU time inside of TC-eligible ops are using tensor cores.
- **Performance summary:**
  - A straightforward panel that shows all of the key metrics from the run in one place

- **Iteration Summary:**
  - A bar chart that shows the amount of time each iteration took during the run. The colored bars are the ones that were used to generate all of the statistics, while the gray bars are iterations that were outside the selected range. Each colored bar shows the breakdown of iteration time into GPU using TC, GPU not using TC, and all other non-GPU time.

- **Top 10 GPU Ops:**
  - Shows the top 10 operations in the run sorted by the amount of GPU time they took. This is a great starting point for trying to find potential for improvements
**System Config:**
- Shows the system configuration for the run.

**Expert Systems Recommendations:**
- Shows any potential problems that DLProf found and recommendations for how to fix them.

**Guidance Panel:**
- Provides some helpful links to learn more about GPU utilization and performance improvements
Detailed views can be found by opening the navigation panel in the upper left corner of the viewer.

All data tables in the detailed views can be sorted, filtered, and exported.
The Op Type Summary view gives a summary of each operation type in the run.
The Ops and Kernels Summaries allows you to drill down into the kernels used by a given operation. When you select an op in the first table, all of the kernels across all instances of that operation will appear in the bottom table.
The **Kernels By Iterations** view allows you to drill down into the kernel instances of an iteration. When you select an iteration in the first table, all of the kernel instances will appear in the bottom table.
The Kernels By Op view allows you to drill down into the kernel instances of a single op instance. First pick an iteration in the top table, and then an op instance in the second table. All kernel instances for that op will appear in the bottom table.
- The **Iterations View** shows a zoomed in iteration bar chart, and gives the user the ability to reaggregate the profiled data using a different iteration range and/or key node values.
• The GPUs View shows a list of all visible GPUs and their utilizations during the run. This view will not exist if the run only used a single GPU.
Use cases
USE CASES
Two different use-cases

Regular users may..
- Want to find the right batch size to increase GPU utilization
- Want to increase Tensor Core usage to reduce training time

Power users may..
- Want to find if there’s regression in their workloads
- Want to know why an Op is taking longer than expected
1. Orthographic Feature Transforms
By Avinash Ahuja
COMPUTER VISION CASE STUDY

- Orthographic Feature Transform for Monocular 3D Object Detection
- ResNet-18 front end
- Academic code - provides opportunities for optimization
  - Example code: [https://github.com/avinashahuja/oft](https://github.com/avinashahuja/oft)
- Runtime stats:
  - ~29 hrs 20 mins to train
  - 600 epochs
  - 8x V100 (16GB) system
    - w/400GB system memory
Add hooks (only for PyTorch)

```python
import nvidia_dlprof_pytorch_nvtx as nvtx
nvtx.init()
```

Wrap the training/inference loop with the PyTorch’s NVTX context manager

```python
with torch.autograd.profiler.emit_nvtx():
    for iter in range(iters):
        # forward pass...
        # backprop...
```

Run dlprof

```
$ dlprof --mode=pytorch python <train script>
```

Analyze results

```
$ tensorboard --logdir ./event_files
```

References:
https://docs.nvidia.com/deeplearning/frameworks/dlprof-user-guide/index.html#quickstart_topic
BASELINE RESULTS

Identifying performance gaps

- Wall clock time is around 15 minutes on an Ampere RTX A6000
- Batch size is 4 and there were 929 mini batches in 1 training epoch (~3700 training points)
- All compute was done in FP32, hence Tensor Core utilization is non-existent.
- GPU Utilization is around 90% which is a good thing.
- DLPprof points out that GPU memory utilization was only 21% and batch size can be increased
- Note: Setting environment variable `NVIDIA_TF32_OVERRIDE=0` disables TF32 so that FP32 is used. It defaults to library and framework settings.
## RESULTS - 1st PASS

Increasing batch size to 16, 20 and 24

<table>
<thead>
<tr>
<th>FP32</th>
<th>BS: 4</th>
<th>BS: 16</th>
<th>BS: 20</th>
<th>BS: 24</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wall Clock Time (min)</td>
<td>14.9</td>
<td>14</td>
<td>13.9</td>
<td>15.3</td>
</tr>
<tr>
<td>Tensor Core Kernel Efficiency (%)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>GPU Utilization (%)</td>
<td>89.8</td>
<td>95</td>
<td>95.2</td>
<td>95.7</td>
</tr>
<tr>
<td>Total Iterations</td>
<td>929</td>
<td>233</td>
<td>187</td>
<td>156</td>
</tr>
<tr>
<td>Profiled Iterations</td>
<td>929</td>
<td>233</td>
<td>187</td>
<td>156</td>
</tr>
<tr>
<td>Start Iteration</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Stop Iteration</td>
<td>928</td>
<td>232</td>
<td>186</td>
<td>155</td>
</tr>
<tr>
<td>Average Iteration Time (s)</td>
<td>0.96</td>
<td>3.59</td>
<td>4.47</td>
<td>5.89</td>
</tr>
<tr>
<td>Memory Utilization (48GB total)</td>
<td>-21%</td>
<td>-71%</td>
<td>-88%</td>
<td>-95%</td>
</tr>
</tbody>
</table>
Tensor Cores are specialized hardware for deep learning that help accelerate matrix multiply and accumulate operations.

NVIDIA Ampere GPUs introduce Tensor Core support for new data types: TF32, Bfloat16, and FP64.

NVIDIA Volta GPUs support using Tensor Cores with FP16 data types.

Deep learning operations that benefit from tensor cores are:

- Fully connected / linear / dense layers
- Convolutional layers
- Recurrent layers
# RESULTS - 2nd PASS

Enabling TF32 - Enabled by default on Ampere Generation devices (TF32 support is from available only on Ampere and future generations)

<table>
<thead>
<tr>
<th>Batch Size</th>
<th>BS: 16</th>
<th>BS: 24</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enabling TF32</td>
<td>FP32</td>
<td>TF32</td>
</tr>
<tr>
<td>Wall Clock Time (min)</td>
<td>14.0</td>
<td>11.1</td>
</tr>
<tr>
<td>Tensor Core Kernel Efficiency (%)</td>
<td>0</td>
<td>47.8</td>
</tr>
<tr>
<td>GPU Utilization (%)</td>
<td>95</td>
<td>93.8</td>
</tr>
<tr>
<td>Total Iterations</td>
<td>233</td>
<td>233</td>
</tr>
<tr>
<td>Profiled Iterations</td>
<td>233</td>
<td>233</td>
</tr>
<tr>
<td>Start Iteration</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Stop Iteration</td>
<td>232</td>
<td>232</td>
</tr>
<tr>
<td>Average Iteration Time (s)</td>
<td>3.59</td>
<td>2.87</td>
</tr>
</tbody>
</table>

Note: Setting environment variable `NVIDIA_TF32.Override=1` or deleting this value enables TF32
AUTOMATIC MIXED PRECISION
Easy to Use and Great Performance

Insert ~two lines of code to use Automatic Mixed-Precision and get up to a 3X speedup

Support for TensorFlow, PyTorch and MXNet

Automatic mixed precision applies two techniques to maximize performance while preserving accuracy:
1) Optimizing per operation precision by casting to FP16 or FP32
2) Dynamic loss scaling to prevent gradient underflow
# RESULTS - 3rd PASS

Enabling Automatic Mixed Precision
Available on Ampere and Volta generation devices

<table>
<thead>
<tr>
<th>With AMP</th>
<th>FP32</th>
<th>AMP</th>
<th>Speedup</th>
<th>FP32</th>
<th>AMP</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPU Generation</td>
<td>Ampere - RTX A6000</td>
<td>Volta - Quadro RTX8000</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Wall Clock Time (min)</td>
<td>14</td>
<td>10.9</td>
<td>22.1%</td>
<td>17.9</td>
<td>15.8</td>
<td>11.7%</td>
</tr>
<tr>
<td>Tensor Core Kernel Efficiency (%)</td>
<td>0</td>
<td>69.2</td>
<td>0</td>
<td>58</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GPU Utilization (%)</td>
<td>95</td>
<td>91.3</td>
<td>96.1</td>
<td>94.1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total Iterations</td>
<td>233</td>
<td>233</td>
<td>233</td>
<td>233</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Profiled Iterations</td>
<td>233</td>
<td>233</td>
<td>233</td>
<td>233</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Start Iteration</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stop Iteration</td>
<td>232</td>
<td>232</td>
<td>232</td>
<td>232</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Average Iteration Time (s)</td>
<td>3.59</td>
<td>2.82</td>
<td>4.6</td>
<td>4.06</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Automatic Mixed Precision on Ampere vs Volta | Batch Size 16
2. AH-Net PyTorch for Clara Healthcare by Michal Marcinkiewicz
# Analyze Your Results

What does DLProf tell me about my workload?

<table>
<thead>
<tr>
<th>Op Name</th>
<th>Op Type</th>
<th>Calls</th>
<th>TC Eligible</th>
<th>Using TC</th>
<th>Kernel Calls</th>
<th>Data Type</th>
<th>Input Shapes</th>
</tr>
</thead>
<tbody>
<tr>
<td>optimizers/grads/UNet_v1/outputs_block/conv2d/Conv2D_grad/Conv2DBackpropFilter</td>
<td>Conv2DBackpropFilter</td>
<td>9</td>
<td>✔</td>
<td>×</td>
<td>18</td>
<td>float32</td>
<td>UNet_v1/outputs_block/concat: &lt;2, 8, 512, 512&gt;, ConstantFolding/optimizers/grads/UNet_v1/grad/Conv2DBackpropFilter</td>
</tr>
<tr>
<td>optimizers/grads/UNet_v1/outputs_block/conv2d_1/Conv2D_grad/Conv2DBackpropFilter</td>
<td>Conv2DBackpropFilter</td>
<td>9</td>
<td>✔</td>
<td>×</td>
<td>18</td>
<td>float32</td>
<td>UNet_v1/outputs_block/act1/relu: &lt;2, 32, 512, 512&gt;, ConstantFolding/optimizers/grads/UNet_v1/grad/Conv2DBackpropFilter</td>
</tr>
<tr>
<td>optimizers/grads/UNet_v1/input_block/conv2d_1/Conv2D_grad/Conv2DBackpropFilter</td>
<td>Conv2DBackpropFilter</td>
<td>9</td>
<td>✔</td>
<td>×</td>
<td>18</td>
<td>float32</td>
<td>UNet_v1/input_block/act1/relu: &lt;2, 32, 512, 512&gt;, ConstantFolding/optimizers/grads/UNet_v1/grad/Conv2DBackpropFilter</td>
</tr>
<tr>
<td>optimizers/grads/UNet_v1/outputs_block/conv2d/Conv2D_grad/Conv2DBackpropInput</td>
<td>Conv2DBackpropInput</td>
<td>9</td>
<td>✔</td>
<td>×</td>
<td>27</td>
<td>float32</td>
<td>ConstantFolding/optimizers/grads/UNet_v1/outputs_block/conv2d/Conv2D_grad/ShapeX-matmul-shapes: &lt;</td>
</tr>
<tr>
<td>UNet_v1/outputs_block/conv2d/Conv2D</td>
<td>Conv2D</td>
<td>9</td>
<td>✔</td>
<td>×</td>
<td>27</td>
<td>float32</td>
<td>UNet_v1/outputs_block/concat: &lt;2, 8, 512, 512&gt;, UNet_v1/outputs_block/conv2d/kernelRead: &lt;3</td>
</tr>
</tbody>
</table>

**GPU Time (ns)**: 18,452,468, CPU Time (ns): 650,629

**Column Descriptions**

- **Op Name**: Specifies the operation name.
- **Op Type**: Indicates the type of operation, e.g., Conv2DBackpropFilter.
- **Calls**: Number of times the operation is called.
- **TC Eligible**: Indicates if the operation is eligible for Tensor Coherence (TC).
- **Using TC**: Indication of whether the operation uses TC.
- **Kernel Calls**: Number of kernel calls.
- **Data Type**: Type of data used in the operation.
- **Input Shapes**: Shapes of the input data.
**ANALYZE YOUR RESULTS**

What does DLProf tell me about my workload?

<table>
<thead>
<tr>
<th>GPU Time (ns)</th>
<th>CPU Time (ns)</th>
<th>Op Name</th>
<th>Op Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>18,452,463</td>
<td>650,629</td>
<td>optimizers/gradients/UNet_v1/outputs_block/conv2d/Conv2D_grad/Conv2DBackpropFilter</td>
<td>Conv2DBackpropFilter</td>
</tr>
<tr>
<td>12,768,639</td>
<td>706,432</td>
<td>optimizers/gradients/UNet_v1/outputs_block/conv2d_1/Conv2D_grad/Conv2DBackpropFilter</td>
<td>Conv2DBackpropFilter</td>
</tr>
<tr>
<td>12,722,109</td>
<td>653,543</td>
<td>optimizers/gradients/UNet_v1/input_block/conv2d_1/Conv2D_grad/Conv2DBackpropFilter</td>
<td>Conv2DBackpropFilter</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>GPU Time (ns)</th>
<th>CPU Time (ns)</th>
<th>Op Name</th>
<th>Op Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>62,815,818</td>
<td>705,566</td>
<td>optimizers/gradients/UNet_v1/outputs_block/conv2d_2/Conv2D_grad/Conv2DBackpropFilter</td>
<td>Conv2DBackpropFilter</td>
</tr>
<tr>
<td>18,421,780</td>
<td>826,402</td>
<td>optimizers/gradients/UNet_v1/outputs_block/conv2d_1/Conv2D_grad/Conv2DBackpropFilter</td>
<td>Conv2DBackpropFilter</td>
</tr>
<tr>
<td>12,779,864</td>
<td>937,740</td>
<td>optimizers/gradients/UNet_v1/inputs_block/conv2d_2d_1/Conv2D_grad/Conv2DBackpropFilter</td>
<td>Conv2DBackpropFilter</td>
</tr>
</tbody>
</table>

cuDNN 8.0.3.13 ➔ cuDNN 8.0.4.12
### EXAMPLE USE CASE
AH-Net PyTorch for CLARA Healthcare

<table>
<thead>
<tr>
<th>GPU Time (ns)</th>
<th>CPU Time (ns)</th>
<th>Op Name</th>
<th>Direction</th>
<th>Op Type</th>
<th>Calls</th>
<th>TC Eligible</th>
<th>Using TC</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,023,657,558</td>
<td>785,062</td>
<td>interpdo</td>
<td>bprop</td>
<td>interpolate</td>
<td>11</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>372,635,580</td>
<td>1,174,081</td>
<td>interpdo</td>
<td>bprop</td>
<td>interpolate</td>
<td>11</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>356,521,002</td>
<td>730,052</td>
<td>interpdo</td>
<td>bprop</td>
<td>interpolate</td>
<td>11</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>214,201,843</td>
<td>8,641,035</td>
<td>conv3d</td>
<td>fprop</td>
<td>conv3d</td>
<td>33</td>
<td>√</td>
<td>×</td>
</tr>
<tr>
<td>199,911,504</td>
<td>15,125,796</td>
<td>interpdo</td>
<td>bprop</td>
<td>conv3d</td>
<td>66</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>173,640,809</td>
<td>16,420,321</td>
<td>interpdo</td>
<td>bprop</td>
<td>conv3d</td>
<td>66</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>163,761,223</td>
<td>8,222,157</td>
<td>interpdo</td>
<td>bprop</td>
<td>interpolate</td>
<td>11</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>152,820,294</td>
<td>1,194,526</td>
<td>interpdo</td>
<td>bprop</td>
<td>interpolate</td>
<td>11</td>
<td>×</td>
<td>×</td>
</tr>
</tbody>
</table>

Top 3 operations are interpolations.

Top 1 operation is 5x slower than the fastest conv3d.
TAKE ACTION
Ensure your findings are acted upon

What to do if you find issues in your model

● Usually issues originate from
  ○ Your code / implementation
  ○ Libraries (Framework, cuDNN, cuBLAS)

● Examples of issues in the code and their fixes
  ○ Change Interpolation to Deconvolution layer
  ○ Use 3x3 kernel instead of 7x7
  ○ Use 32 filters instead of 30 for convolutions

● Issues in libraries
  ○ Framework - create example and open a bug
  ○ cuDNN - gather logs and open a bug
  ○ cuBLAS - gather logs and open a bug
# EXAMPLE USE CASE

## AH-Net PyTorch for CLARA Healthcare

Replacing interpolation by deconvolution removes the bottleneck.
REFERENCE

User Guide


Profiling with Deep Learning Profiler

https://docs.nvidia.com/deeplearning/frameworks/index.html#profiling-with-dlprof

Demo - How to use DLPprof with PyTorch

https://docs.google.com/presentation/d/1J8lzrb9F5NWoqzMkjG3fEljYYfraHrLuYtNaM7vTeAs/edit?usp=s haring

GTC 2021 - Deep Learning Performance Optimization with Profiling Tools [S31228]

https://gtc21.event.nvidia.com/media/Deep%20Learning%20Performance%20Optimization%20with%20Pr ofiling%20Tools%20%5BS31228%5D/1_o8wx0hs0
THANK YOU