Profiling Deep Learning Workloads

Murali Emani,
Data Science group, ALCF
memani@anl.gov
Introduction

• Profiling is an approach to measure application performance

• Simple Profiling:
  - How long does an application take

• Advanced Profiling:
  - Why does an operation take long time

• Goal: Find performance bottlenecks
  - inefficient programming
  - memory I/O bottlenecks
  - parallel scaling
Typical Optimization Workflow

Profile application

Inspect and analyze

Optimize

Iterative workflow till desired performance is reached
Broad classification

- Hardware counters
  count events from CPU perspective (# of flops, memory loads, etc.)
  usually needs Linux kernel module installed or root permission

- Statistical profilers (sampling)
  interrupt program at given intervals to find the state of a program

- Event based profilers (tracing)
  collect information on each function call
Plethora of Tools

- Cprofile
- Gprof
- Perf tool
- Intel Vtune
- HPCToolKit
- OpenSpeedShop
- TAU
- Nvidia Nvprof, Nsight

...
Profiling DNN workloads

• Critical to understand workload performance

• Machine learning and deep learning models are implemented on a variety of hardware

• Most applications are written in Python using standard ML frameworks

• The frameworks generate kernels based on hardware and customized installation and libraries (MKL-DNN, CuDNN etc.)
Challenges

• Profiling is hard, cumbersome and time-consuming

• Profiling tools generate lot of data and hard to understand

• The problem is further compounded with large, complex models with large volumes of data

• Need strategies to use right tools and detailed insights to how to analyze the profile data
Profiling on Nvidia GPUs
Profiling on Nvidia GPUs

Use Nvidia profiler ‘Nvprof’
- capture metrics from hardware counters
- invoked via command line or UI (Nvidia Visual Profiler NVVP)

See list of options using

```
nvprof -h
```

Some useful options:
- `-o`: create output file to import into nvvp
- `--metrics` / `-m`: collect metrics
- `--events` / `-e`: collect events
- `--log-file`: create human readable output file
- `--analysis-metrics`: collect all metrics to import into nvvp
- `--query-metrics`/`--query-events`: list of available metrics/events
Events and Metrics

• An **event** is a countable activity, action, or occurrence on a device. It corresponds to a single hardware counter value which is collected during kernel execution.

• A **metric** is a characteristic of an application that is calculated from one or more event values.

  *In general, events are only for experts, rarely used.*

• Vary in number based on hardware family (P100, K80, V100 etc).

• For example, on V100, nvprof gives 175 metrics.

• Event and metric values are aggregated across all units in the GPU.
Workflow – on Cooley

Option 1)
- Use ‘nvprof’ to collect metrics in an output file (compute node)
- Use ‘nvvp’ to visualize the profile (login node)

Option 2)
- Directly launch nvvp on compute node and profile the code interactively
Profile Commands

- Kernel timing analysis:
  
  ```
  nvprof --log-file timing.log <myapp>
  nvprof --log-file timing.log python myapp.py args
  ```

- Traces (#threads, #warps, #registers)
  
  ```
  nvprof --print-gpu-traces --log-file traces.log <myapp>
  ```

- Get all metrics for all kernels
  
  ```
  nvprof --metrics all --log-file all-metrics.log <myapp>
  ```

- Get metrics for guided analysis
  
  ```
  nvprof --analysis-metrics -o analysis.nvvp <myapp>
  ```

- Visual profile to use Nvidia Visual Profiler (nvvp)
  
  ```
  nvprof -o analysis.nvvp <myapp>
  ```
Selective Profiling

- As profiling adds significant overhead, a better strategy is to profile only regions of interest (kernels and metrics).

- All metrics for kernels of interest:
  ```
  nvprof --profile-from-start off --kernels <kernel-name> --metrics all --log-file selective-profile.log <myapp>
  ```

- Few metrics for kernels of interest
  ```
  nvprof --profile-from-start off --kernels <kernel-name> --metrics ipc --log-file selective-profile.log <myapp>
  ```

For example, if we want to profile heavy kernels only

  1) use nvprof to list all kernels sorted by the time
  2) re-run nvprof in selective profiling mode

- Profile GEMM kernels
  ```
  nvprof --profile-from-start off --kernels "::gemm:1" --metrics all --log-file selective-profile.log <myapp>
  ```
## Metrics and Events

Metrics relevant to identify compute, memory, IO characteristics

<table>
<thead>
<tr>
<th>** Metric</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>achieved_occupancy</td>
<td>Ratio of the average active warps per active cycle to the maximum number of warps supported on a multiprocessor</td>
</tr>
<tr>
<td>ipc</td>
<td>Instructions executed per cycle</td>
</tr>
<tr>
<td>gld_efficiency</td>
<td>Ratio of requested global memory load throughput to required global memory load throughput expressed as percentage.</td>
</tr>
<tr>
<td>gst_efficiency</td>
<td>Ratio of requested global memory store throughput to required global memory store throughput expressed as percentage.</td>
</tr>
<tr>
<td>dram_utilization</td>
<td>The utilization level of the device memory relative to the peak utilization on a scale of 0 to 10</td>
</tr>
</tbody>
</table>
Detailed Analysis

Use visual profiler nvvp

The guided analysis system walks you through the various analysis stages to help you understand the optimization opportunities in your application. Once you become familiar with the optimization process, you can explore the individual analysis stages in an unguided mode. When optimizing your application it is important to fully utilize the compute and data movement capabilities of the GPU. To do this you should look at your application’s overall GPU usage as well as the performance of individual kernels.

Examine GPU Usage
Determine your application’s overall GPU usage. This analysis requires an application timeline, so your application will be run once to collect it if it is not already available.

Examine Individual Kernels
Determine which kernels are the most performance critical and that have the most opportunity for improvement. This analysis requires utilization data from every kernel, so your application will be run once to collect that data if it is not already available.

Delete Existing Analysis Information
If the application has changed since the last analysis then the existing analysis information may be stale and should be deleted before continuing.

Switch to unguided analysis
## Kernel Optimization Priorities

The following kernels are ordered by optimization importance based on execution time and achieved occupancy. Optimization of higher ranked kernels (those that appear first in the list) is more likely to improve performance compared to lower ranked kernels.

<table>
<thead>
<tr>
<th>Rank</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>[ 2 kernel instances ] maxwell_sgemm_128x64_tn</td>
</tr>
<tr>
<td>1</td>
<td>[ 1 kernel instances ] elementWise(float*, float*, float*, float*, float*, float*)</td>
</tr>
</tbody>
</table>
Example

Simple CNN in Keras

```python
model = Sequential()
model.add(Conv2D(32, kernel_size=(3, 3), activation='relu',
                 input_shape=input_shape))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(num_classes, activation='softmax'))

model.compile(......)

model.fit(.....)
```

https://github.com/keras-team/keras/blob/master/examples/mnist_cnn.py
Profiling result:

<table>
<thead>
<tr>
<th>Type</th>
<th>Time(%)</th>
<th>Time</th>
<th>Calls</th>
<th>Avg</th>
<th>Min</th>
<th>Max</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPU activities:</td>
<td>36.04%</td>
<td>46.6474s</td>
<td>202752</td>
<td>230.07us</td>
<td>124.57us</td>
<td>339.61us</td>
<td>void sgemm_largek_lds64&lt;bool=0, bool, int, int, int, int, int, float const <em>, float const <em>, float, float, int, int, int</em>, int</em>)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4.90% 5.34396s 1.1244ms 52.928us 1.2132ms void fermiPlusCgemmLDS128_batched&lt;float const <em>, float2</em> const <em>, float2</em> const <em>, float2</em>, float2 const *, float2 const *, int, int, int, int, int, int, int&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4.60% 5.95521s 907.53us 252.41us 1.4053ms cgemm_strided_batched_sm35_ldg_nt_64 int, int, int, int, int, float const <em>, float const <em>, float, float, int, int, int</em>, int</em>)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4.22% 5.46673s 13778 396.77us 34.912us 536.64us void tensorflow::BiasNCHWKernel&lt;float const *, float const <em>, float const <em>, float const, float</em>, float</em>, int, int, int&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3.44% 4.44991s 6889 645.94us 185.95us 748.12us void sgemm_largek_lds64&lt;bool=0, bool, int, int, int, int, float const <em>, float const <em>, float, float, int, int, int</em>, int</em>)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3.35% 4.33847s 12200 355.61us 43.871us 473.31us void fft2d_c2r_32x32&lt;float, bool=0, bool, int, int, int, float, float, cudnn::reduced_divisor, bool, float*, float*, int2, int, int&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3.19% 4.12895s 12208 338.22us 27.552us 464.35us void fft2d_r2c_32x32&lt;float, bool=0, bool, int2, int, int&gt;</td>
</tr>
</tbody>
</table>

API calls: 47.93% 42.3065s 6891 6.1394ms 4.2120us 8.8403ms cuCtxSynchronize
26.73% 23.5916s 2092524 11.274us 4.8920us 40.285ms cudaLaunchKernel
5.37% 4.74056s 8 592.57ms 1.9750us 4.74054s cudaStreamCreateWithFlags
5.32% 4.69823s 209098 22.469us 545ns 4.29630s cudaPointerGetAttributes
3.81% 3.36602s 356228 9.4490us 4.1710us 34.767ms cudaMempcpyAsync
2.37% 2.09162s 231818 9.0220us 358ns 9.9905ms cuEventRecord
1.37% 1.20962s 66998 18.054us 5.8700us 6.1906ms cuMempcpyHtoDAsync
1.29% 1.14020s 833508 1.3670us 438ns 2.1696ms cuEventQuery
0.88% 0.85745ms 37551 33.100us 6.4060us 7.6150ms cuMempcpyDtoHAsync
Nvidia Nsight Tools

• **Nsight Systems** - System-wide application algorithm tuning
• **Nsight Compute** – Debug CUDA API and optimize CUDA kernels

• To profile
  
  $ nsys profile python train.py

• This generates profile file in ‘report.qdrep’ which can be imported to view with Nsight Systems UI

• To identify which kernels are run on Tensorcores (dedicated HW units for half/mixed precision matrix multiply-accumulate ops)
  
  $ nv-nsight-cu-cli --kernel-id ::s884:1 python train.py
NSIGHT SYSTEMS
Next-Gen System Profiling Tool

System-wide application algorithm tuning
Multi-process tree support

Locate optimization opportunities
Visualize millions of events on a fast GUI timeline
Or gaps of unused CPU and GPU time

Balance your workload across multiple CPUs and GPUs
CPU algorithms, utilization, and thread state
GPU streams, kernels, memory transfers, etc

Multi-platform: Linux & Windows, x86-64 & Tegra, MacOSX (host only)

Processes and threads
CUDA and OpenGL API trace
cuDNN and cuBLAS trace
Kernel and memory transfer activities
Multi-GPU

https://bluewaters.ncsa.illinois.edu/liferay/content/document-library/content/NVIDIA%20Nsight%20Systems%20Overview%20by%20Sneha%20Kottapalli.pdf
NSIGHT COMPUTE
Next-Gen Kernel Profiling Tool

Key Features:
- Interactive CUDA API debugging and kernel profiling
- Fast Data Collection
- Improved Workflow (diffing results)
- Fully Customizable (programmable UI/Rules)
- Command Line, Standalone, IDE Integration

OS: Linux, Windows, ARM, MacOSX (host only)
GPUs: Pascal (GP10x), Volta, Turing

https://bluewaters.ncsa.illinois.edu/liferay/content/document-library/content/NVIDIA%20Nsight%20Systems%20Overview%20by%20Sneha%20Kottapalli.pdf
Profiling on CPUs using Intel Vtune
Application Performance Snapshot (APS)

APS generates a highlevel performance snapshot of your application. Easy to run:

```bash
source /soft/compilers/intel/19.0.3.199/vtune_amplifier/apsvars.sh
export
LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/soft/compilers/intel/19.0.3.199/vtune_amplifier/lib64
aps --result-dir=aps_results/ -- python /full/path/to/script.py
```

Results can be viewed in a single html file, or via command line:

<table>
<thead>
<tr>
<th>Summary information</th>
</tr>
</thead>
<tbody>
<tr>
<td>HW Platform: Intel(R) Processor code named Knights Landing</td>
</tr>
<tr>
<td>Logical core count per node: 256</td>
</tr>
<tr>
<td>Collector type: Driverless Perf system-wide counting</td>
</tr>
<tr>
<td>Used statistics: aps_results</td>
</tr>
</tbody>
</table>

Your application might underutilize the available logical CPU cores because of insufficient parallel work, blocking on synchronization, or too much I/O. Perform function or source line-level profiling with tools like Intel(R) VTune(TM) Amplifier to discover why the CPU is underutilized.

CPU Utilization: 6.50%

Your application might underutilize the available logical CPU cores because of insufficient parallel work, blocking on synchronization, or too much I/O. Perform function or source line-level profiling with tools like Intel(R)
Application Performance Snapshot (APS)

Pros

- Very easy to use
- Tracks important hardware metrics:
  - Thread Load Balancing
  - Vectorization
  - CPU Usage

Cons

- Only high level information – but then again, that is the design of this tool.
Intel Vtune – Hotspots

```
sampling-mode=sw - User-Mode Sampling (default) used for profiling:
  • Targets running longer than a few seconds
  • A single process or a process-tree
  • Python and Intel runtimes

sampling-mode=hw - (Advanced hotspots) Hardware Event-Based Sampling used for profiling:
  • Targets running less than a few seconds
  • All processes on a system, including the kernel
```
Intel Vtune – Advanced Hotspots

Advanced Hotspots analysis

- Detailed report of how effective the computation is on CPUs
- extends the hotspots analysis by collecting call stacks, context switch and statistical call count data and analyzing the CPI (Cycles Per Instruction) metric.

```
amplxe-cl -collect hotspots -knob sampling-mode=hw -finalization-mode=none -r vtune-result-dir_dir_advancedhotspots/ -- python /full/path/to/script.py
```

Run the finalization step after the run completes from the login nodes

```
amplxe-cl -finalize -search-dir / -r vtune-result-dir_dir_advancedhotspots
```
Intel Vtune – Advanced Hotspots

Run the GUI to view your results:

```
amplxe-gui vtune-result-dir_advancedhotspots
```
Useful Commands

amplxe-cl -c hotspots -- python3 myapp.py
amplxe-cl -R hotspots -report-output report-hotspots.csv -format csv

amplxe-cl -c uarch-exploration -k sampling-interval=100 -- python3 myapp.py
amplxe-cl -R uarch-exploration -report-output report-uarch-exploration.csv -format csv

amplxe-cl -c memory-access -k sampling-interval=100 -- python3 myapp.py
amplxe-cl -R memory-access -report-output report-memory-access.csv -format csv

amplxe-cl -c memory-consumption -k sampling-interval=100 -- python3 myapp.py
amplxe-cl -R memory-consumption -report-output report-memory-consumption.csv -format csv

change sampling interval
-k sampling-interval=<number>
Useful Commands

amplxe-cl -report hw-events/summary -r r000ue/ -report-output ./report-uarch.csv -format csv

amplxe-cl -collect hotspots -strategy ldconfig:notrace:notrace -- python myapp.py

## get MKL-DNN verbose
export MKLDNN_VERBOSE=2
amplxe-cl -collect hotspots -strategy ldconfig:notrace:notrace -- python myapp.py
Hands-on Exercise

Example scripts to profile an image classification CNN model with TF/Keras

https://github.com/argonne-lcf/ATPESC_MachineLearning

cd Profiling

Cooley
qsub -A training -q training -t 1:00:00 -n 1 qsub_mnist_profile_gpu.sh

Theta
qsub -A ATPSEC2020 -q ATPSEC2020 -t 1:00:00 -n 1 qsub_mnist_profile_cpu.sh
Thank you!
backup
Operations on backward weights, data have stalls $\rightarrow$ high memory requirements
- Convolution layer is sensitive to compute units, memory and cachelines
- Dense layer is sensitive to communication -> bandwidth
VTune profiling

source /opt/intel/vtune_amplifier/amplxe-vars.sh
aprun -n ... -e OMP_NUM_THREADS=128 \
-e LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/opt/intel/vtune_amplifier/lib64 \
ampxle-cl -collect advance-hotspots -r output_dir python script.py

The python modules are compiled using -g flag. Therefore, the user could trace the source file in Vtune.

More details: Profiling Your Application with Intel VTune and Advisor - Carlos Rosales-Fernandez and Paulius Velesko, Intel

Remember to set LD_LIBRARY_PATH, Put vtune library at the end!! Otherwise, it might complaint about the GLIBCXX version.
GPU Memory - metrics

1. dram_read_throughput, dram_read_transactions
2. dram_write_throughput, dram_write_transactions
3. sysmem_read_throughput, sysmem_read_transactions
4. sysmem_write_throughput, sysmem_write_transactions
5. l2_l1_read_transactions, l2_l1_read_throughput
6. l2_l1_write_transactions, l2_l1_write_throughput
7. l2_tex_read_transactions, l2_texture_read_throughput
8. texture is read-only, there are no transactions possible on this path
9. shared_load_throughput, shared_load_transactions
10. shared_store_throughput, shared_store_transactions
11. l1_cache_local_hit_rate
12. l1 is write-through cache, so there are no (independent metrics for this path - refer to other local metrics
13. l1_cache_global_hit_rate
14. see note on 12
15. gld_efficiency, gld_throughput, gld_transactions
16. gst_efficiency, gst_throughput, gst_transactions

https://stackoverflow.com/questions/37732735/nvprof-option-for-bandwidth
GEMM – $2 \times m \times n \times k$ operations
$m, k$ – hidden layer size
$n$ = minibatch size

$$2 \times 512 \times 512 \times 64 = 0.03 \text{ GFLOP}$$

Peak upper limit = 6000 GFLOP/s

Runtime ~ 5.6 usec

<table>
<thead>
<tr>
<th>Time(%)</th>
<th>Time</th>
<th>Calls</th>
<th>Avg</th>
<th>Min</th>
<th>Max</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>93.93%</td>
<td>575.72us</td>
<td>8</td>
<td>71.96us</td>
<td>70.241us</td>
<td>78.945us</td>
<td>maxwell_sgemm_128x64_tn</td>
</tr>
</tbody>
</table>
Optimization

\[
\begin{align*}
[A_1][h] &= [x_1] \\
[A_2][h] &= [x_2] \\
[A_3][h] &= [x_3] \\
[A_4][h] &= [x_4]
\end{align*}
\]

Combined Matrices

**SGEMM Performance Improvement #1**

- As our matrix operations share inputs we can combine them

<table>
<thead>
<tr>
<th>Name</th>
<th>Time(%)</th>
<th>Time</th>
<th>Calls</th>
<th>Avg</th>
<th>Min</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>maxwell_sgemm_128x64_tn</td>
<td>93.93%</td>
<td>575.72us</td>
<td>8</td>
<td>71.96us</td>
<td>70.24us</td>
<td>78.94us</td>
</tr>
<tr>
<td>maxwell_sgemm_128x64_tn</td>
<td>84.40%</td>
<td>198.11us</td>
<td>2</td>
<td>99.06us</td>
<td>98.18us</td>
<td>99.94us</td>
</tr>
</tbody>
</table>

- From ~500 GFLOP/s to ~1350 GFLOP/s
- 2.5x performance gain