Ministry of Electronics & Information Technology

सुस्वागतम्
நல்வரவு
ಸುಸ್ವಾಗತಂ
സുസ്വാഗതം
ସୁସ୍ୱାଗତମ
ਸੁਆਗਤਮ
خوش آمدید

NPSF User Workshop 2023

24 July – 24 Aug

Presentation On

Muti-GPU and Multi-node Training and Profiling

Abhishek Choudhary– Quantiphi

25th July

National PARAM Supercomputing Facility

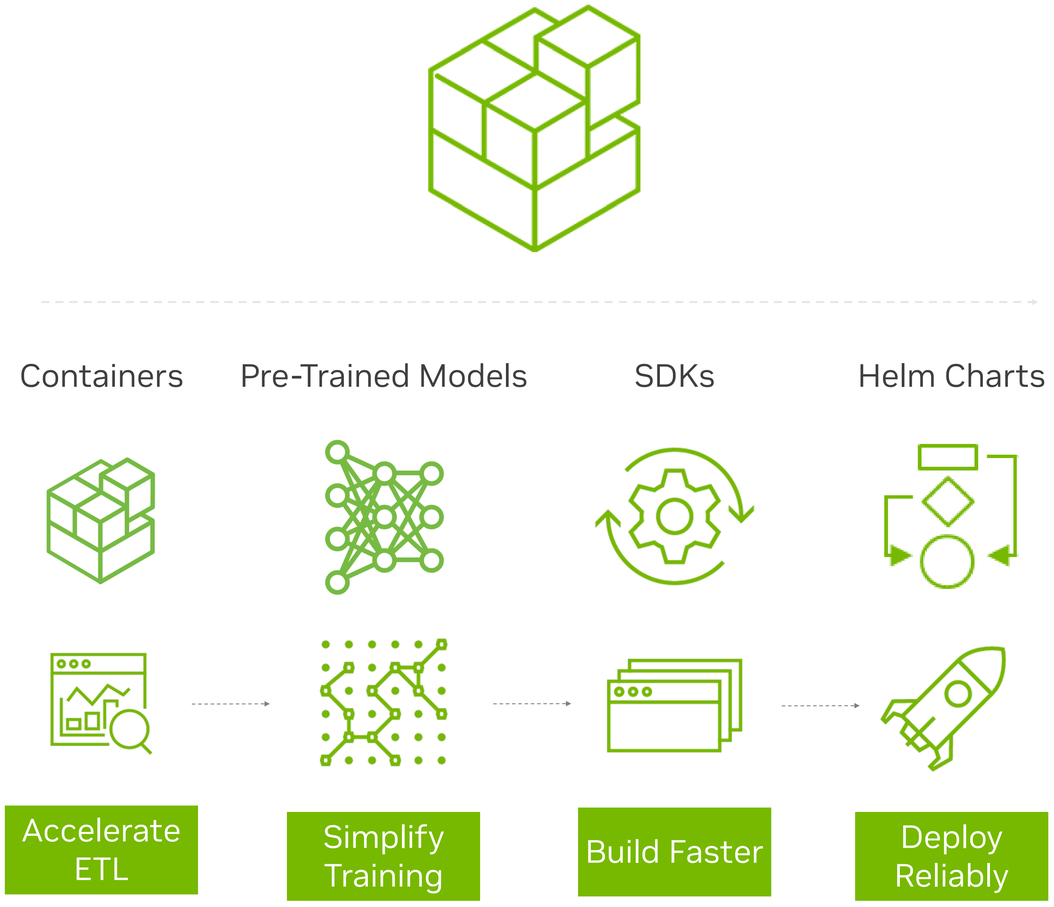
Agenda

- **NVIDIA NGC**
  - NGC Overview, Container Environment, Using NGC
- **SLURM**
- **Enroot and Job Scheduling**
  - Issues with Docker, Enroot Overview, Pyxis, Job Scheduling
- **Job Scheduling Best Practices**
- **Profiling AI/HPC Jobs**
  - Tips and Techniques for Efficient Workloads
- **Resources**

NVIDIA NGC Overview

NVIDIA NGC: AI BUILDING BLOCKS

Faster time to insight



NGC Containers

- Include all necessary libraries
- Performance Optimized | Updated Monthly | Secure | Reliable

Pre-Trained Models

- Best Model starting point created by NVIDIA experts
- Faster Training | Higher Accuracy | Credential Transparency

SDKs

- Easy development. Build End-to-end AI Faster

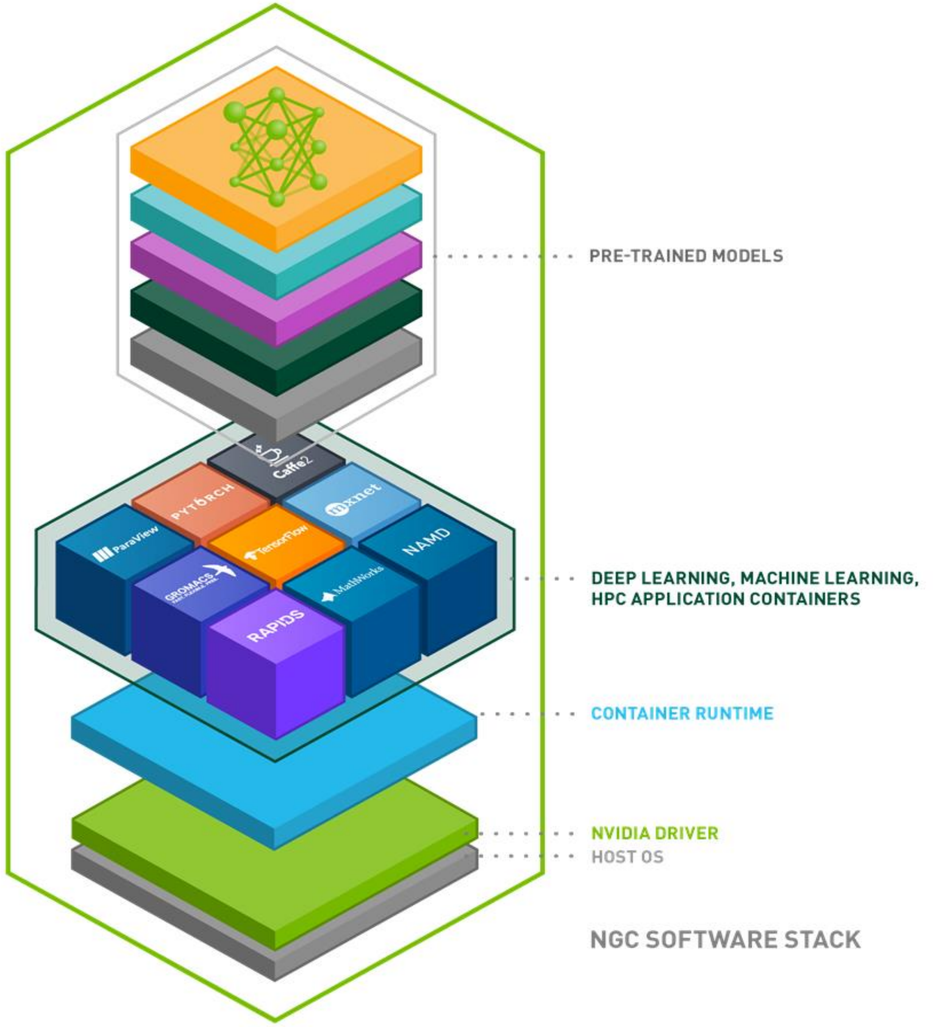
Helm Charts

- Automatic and reliable deployments

CONTAINER ENVIRONMENT

Overview

- A container is a portable unit of software that combines the application and all its dependencies into a single package that's agnostic to the underlying host OS.
- It removes the need to build complex environments and simplifies the application development-to-deployment process.
- The **NVIDIA NGC catalog** contains a host of GPU-optimized containers for deep learning, machine learning, visualization, and high-performance computing (HPC) applications that are tested for performance, security, and scalability.
- **Benefits**:
  - Deploy Easily
  - Train Faster
  - Run Anywhere
  - Deploy with Confidence



Complete Software Stack to build AI

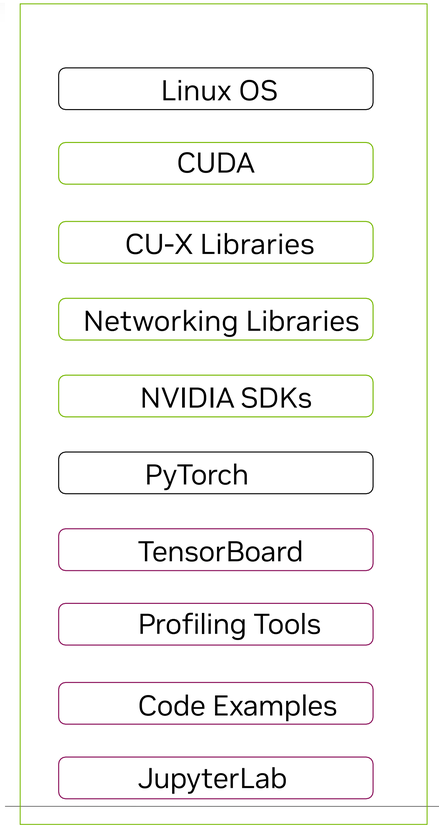
Contents of the PyTorch container

- The complete source of the NVIDIA version of PyTorch.
- Pre-built and installed as a system Python module.
- Sample scripts that demonstrates efficient training of convolutional neural networks (CNNs).

Contents of the PyTorch container

This container image contains the complete source of the version of PyTorch in `/opt/pytorch`. It is prebuilt and installed in the default Python environment (`/usr/local/lib/python3.8/dist-packages/torch`) in the container image. The container also includes the following:

- Ubuntu 20.04 including Python 3.8
- NVIDIA CUDA® 12.1.0
- NVIDIA cuBLAS from CUDA 12.1.0
- NVIDIA cuDNN 8.8.1.3
- NVIDIA NCCL 2.17.1
- NVIDIA RAPIDS™ 23.02
- Apex
- rdma-core 36.0
- NVIDIA HPC-X 2.13
- OpenMPI 4.1.4+
- GDRCopy 2.3
- TensorBoard 2.9.0
- Nsight Compute 2023.1.0.15
- Nsight Systems 2023.1.1.127
- NVIDIA TensorRT™ 8.5.3
- Torch-TensorRT 1.4.0dev0
- NVIDIA DALI® 1.23.0
- MAGMA 2.6.2
- JupyterLab 2.3.2 including Jupyter-TensorBoard
- TransformerEngine 0.6.0
- PyTorch quantization wheel 2.1.2



- Each NGC container contains all the necessary software for you to start developing your AI models
- AI Frameworks: TensorFlow or PyTorch
- Inside the PyTorch NGC container:
- Container Link: https://catalog.ngc.nvidia.com/orgs/nvidia/containers/pytorch
- Release Notes: https://docs.nvidia.com/deeplearning/frameworks/pytorch-release-notes/index.html
- Support Matrix: https://docs.nvidia.com/deeplearning/frameworks/support-matrix/index.html

USING NGC CONTAINERS

Containers

Main interface for Containers

- Locate containers by keyword
- One click shortcut to docker pull command
- Follow run instructions

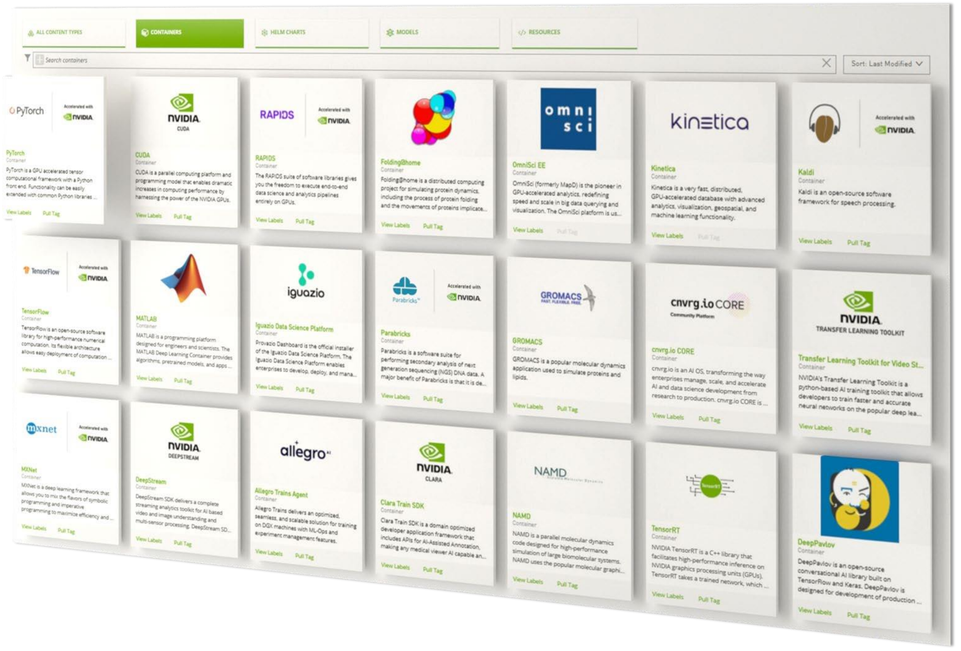
Steps to run TensorFlow:

1. *docker pull nvcr.io/nvidia/tensorflow:22.03-tf1-py3*
2. *docker run -it --rm -v local_dir:container_dir nvcr.io/nvidia/tensorflow:xx.xx-tfx-py3*

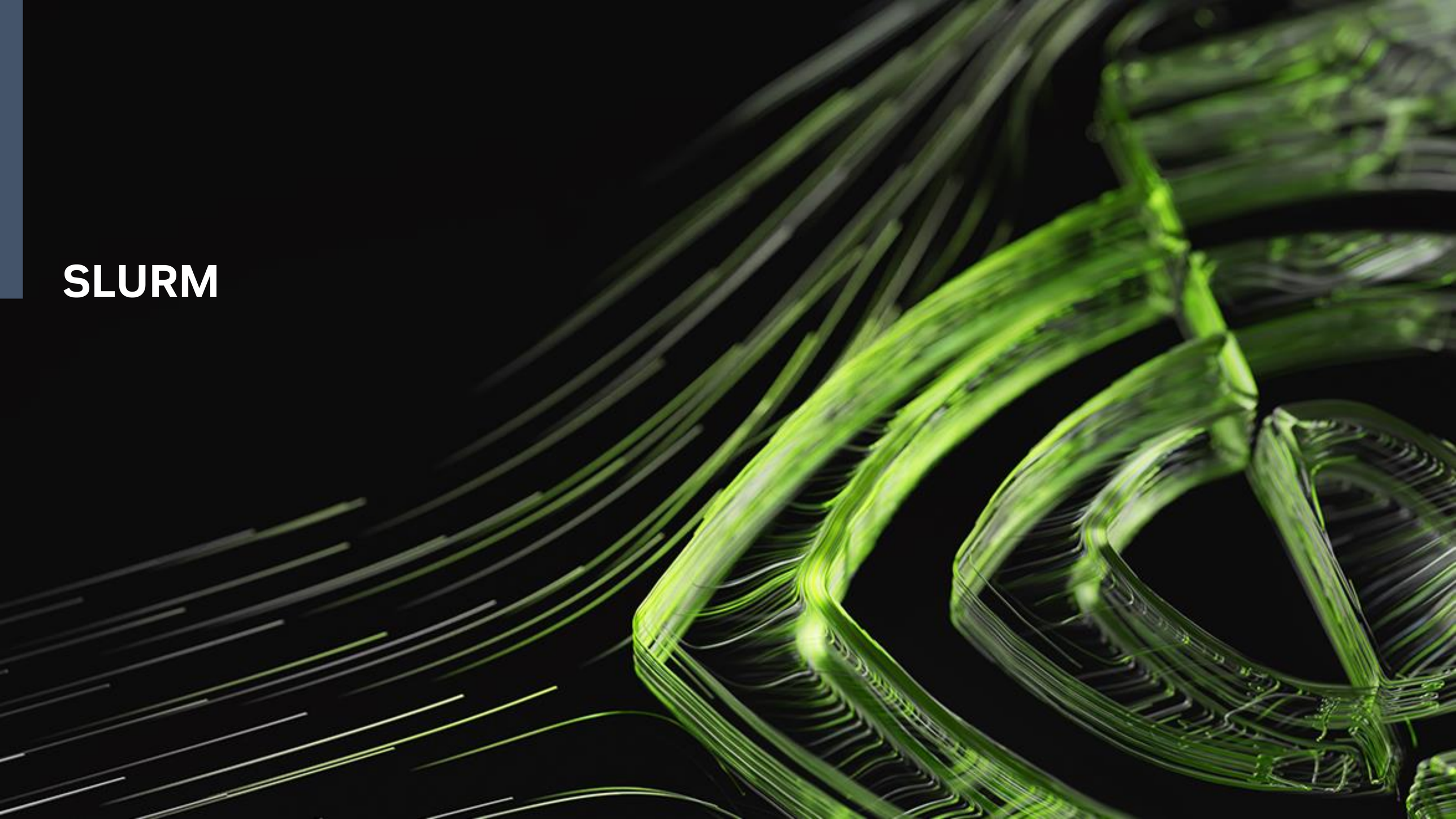
Build your own Application with your custom container - Dockerfile:

FROM nvcr.io/nvidia/tensorflow:22.02-tf1-py3

NGC Containers: ngc.nvidia.com/catalog/containers

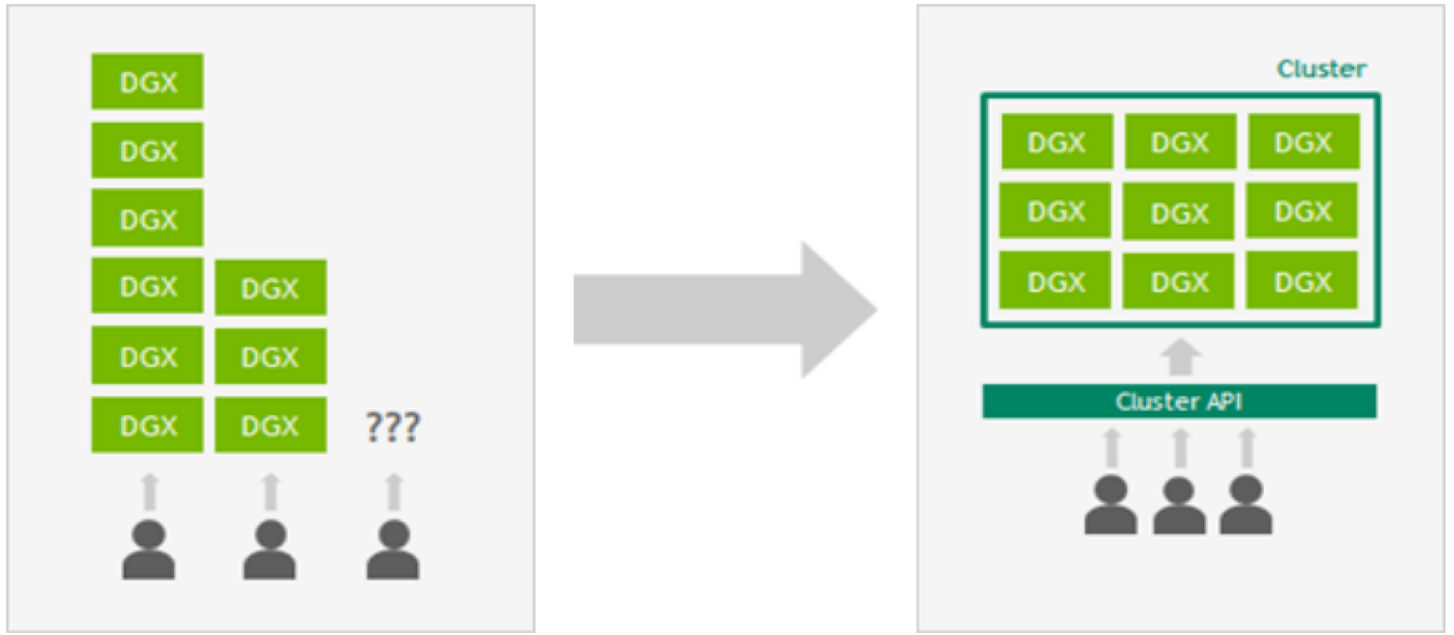


SLURM



Orchestration with SLURM

Why Orchestration?
Manage multi-node resources more efficiently, at scale



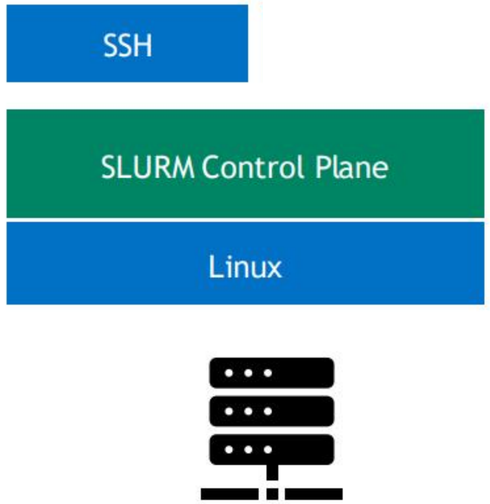
Dedicated Nodes:

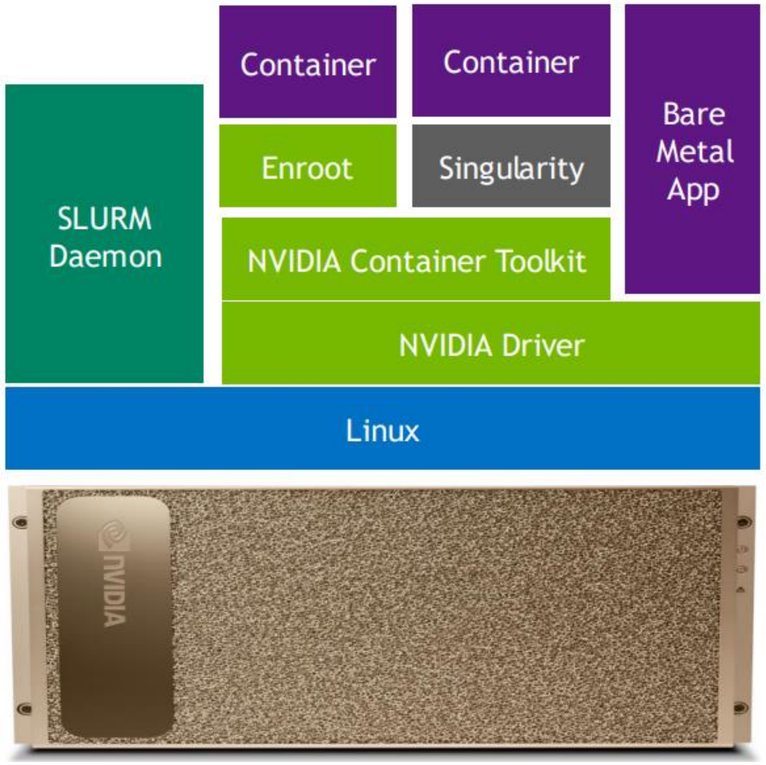
- User coordination required
- Resources can go unused
- Not scalable
- Lack of maintainability

Cluster:

- Less user coordination / Automatic Job Scheduling
- Efficient use of resources
- Highly scalable
- Standardized maintenance

Typical SLURM Software Stack





Enroot and Job Scheduling

Issues With Using Docker

- Security. If users are in the docker group, then they can trivially get root access on the node.

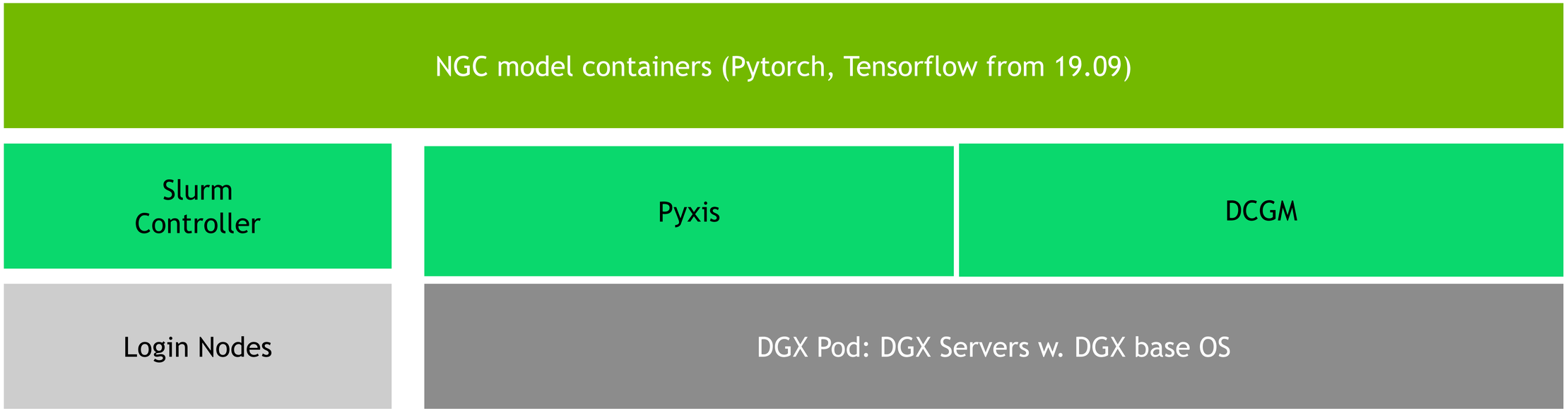
- System Administration. Even with trusted users, root privileges cause issues which require
- sysadmin intervention (e.g. file permissions on NFS, etc.)

- Resource isolation. Slurm can run tasks inside of cgroups which limit the resources for the
task (CPU, GPU, RAM, etc.). But when a user starts a docker container, they escape the
cgroups and lose the isolation.

- Usability. Docker is pretty well-known at this point, but there is still a learning curve for new users.

NVIDIA Enroot

NVIDIA open-source tool to convert traditional container/OS images into unprivileged sandboxes



- Fully unprivileged “chroot”
- Standalone
- Little isolation, no overhead
- Docker image support
- Simple image format
- Composable and extensible
- Simple and easy to use

Enroot Basic Usage

- SSH on the login node
- Enroot Import : Import the Pytorch image from NVIDIA GPU Cloud/$

```
$ enroot import 'docker://nvcr.io#nvidia/pytorch/manifests/23.06-py3'
```

- Enroot Create: Create a container from the sqsh file

```
$ enroot create –name pytorch nvidia+pytorch+23.06-py3.sqsh
```

- Enroot List: List the containers

```
$ enroot list
pytorch
```

- Enroot Start: Run a script inside a container . --rw flag for enabling read write permissions inside the container , --mount flag for mounting a directory inside the container $ enroot import 'docker://$

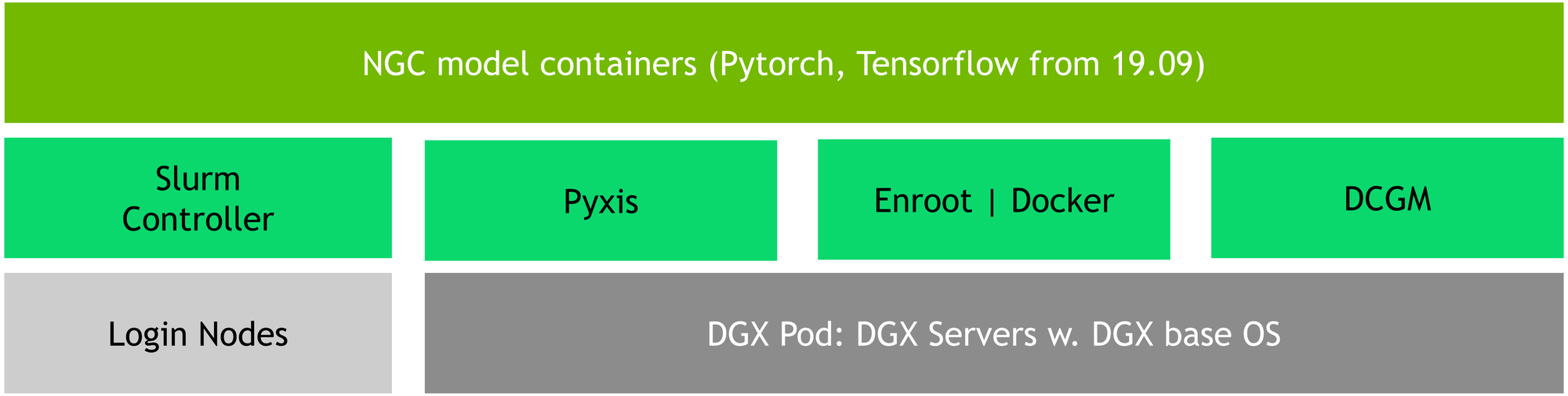
```
$ enroot start –-rw pytorch –mount /home/examples/scripts/:/workspace/ sh -c "python3 main.py"
```

- Enroot Remove: Remove the containerport 'docker://$

```
$ enroot remove pytorch
```

Pyxis: The SLURM Plugin for ENROOT

NVIDIA open-source tool to convert traditional container/OS images into unprivileged sandboxes



- Pyxis is a SPANK plugin for the Slurm Workload Manager. It allows unprivileged cluster users to run containerized tasks through the srun command.
- Seamlessly execute the user's task in an unprivileged container.
- Simple command-line interface.
- Fast Docker image download with support for layers caching and layers sharing across users.
- Supports multi-node MPI jobs through PMI2 or PMIx (requires Slurm support).
- Allows users to install packages inside the container.
- Works with shared filesystems.
- Does not require cluster-wide management of subordinate user/group ids.

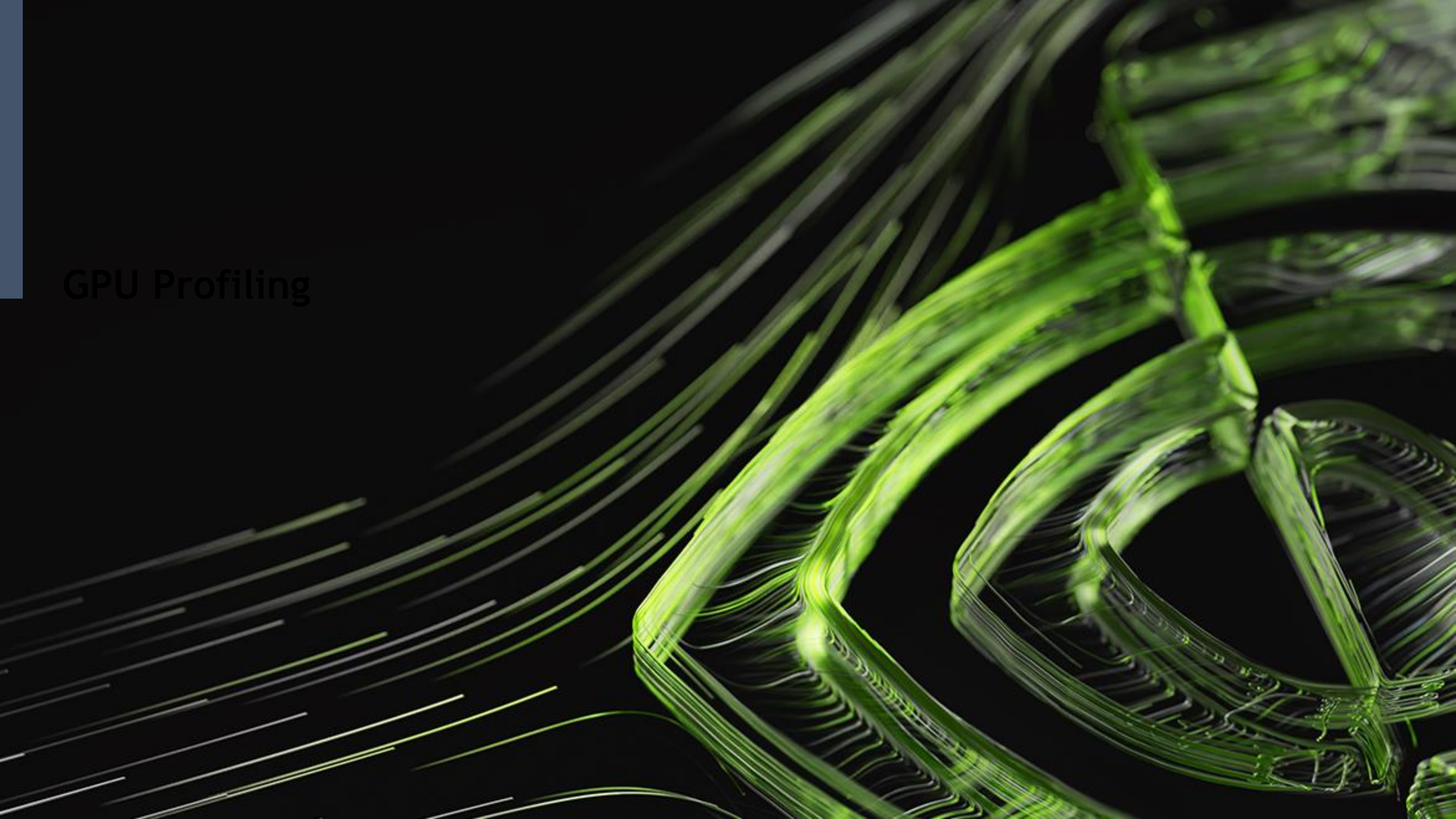
SLURM Example 2 - Batch Jobs

```
#!/bin/bash


#SBATCH --job-name=singlenode-example
#SBATCH --nodes=1
#SBATCH --partition=dgxnp
#SBATCH --ntasks=1
#SBATCH –cpus-per-task=4
#SBATCH --gres=gpu:A100-SXM4:8




nodes=( $( scontrol show hostnames $SLURM_JOB_NODELIST ) )
nodes_array=($nodes)
head_node=${nodes_array[0]}
echo $head_node
head_node_ip=$(srun --nodes=1 --ntasks=1 -w "$head_node" hostname --ip-address)
echo $head_node_ip
echo Node IP: $head_node_ip
export LOGLEVEL=INFO
head_node_array=($head_node_ip)
head_n=${head_node_array[1]}


srun --no-container-entrypoint  --container-image $(pwd)/quantiphinvidiapractice+pytorch+23.05-py3.sqsh --container-mounts $(pwd)/minGPT-ddp/:/workspace/ \
torchrun \
--nnodes 1 \
--nproc_per_node 8 \
--rdzv_id $RANDOM \
--rdzv_backend c10d \
--rdzv_endpoint $head_n:29500 \
/workspace/mingpt/main.py
```

SLURM Example 3 - Batch Multi-node jobs

```
#!/bin/bash


#SBATCH --job-name=multinode-example
#SBATCH --nodes=2
#SBATCH --partition=dgxnp
#SBATCH --ntasks=2
#SBATCH –cpus-per-task=4
#SBATCH --gres=gpu:A100-SXM4:8




nodes=( $( scontrol show hostnames $SLURM_JOB_NODELIST ) )
nodes_array=($nodes)
head_node=${nodes_array[0]}
echo $head_node
head_node_ip=$(srun --nodes=1 --ntasks=1 -w "$head_node" hostname --ip-address)
echo $head_node_ip
echo Node IP: $head_node_ip
export LOGLEVEL=INFO
head_node_array=($head_node_ip)
head_n=${head_node_array[1]}


srun --no-container-entrypoint  --container-image $(pwd)/quantiphinvidiapractice+pytorch+23.05-py3.sqsh --container-mounts $(pwd)/minGPT-ddp/:/workspace/ \
torchrun \
--nnodes 2 \
--nproc_per_node 8 \
--rdzv_id $RANDOM \
--rdzv_backend c10d \
--rdzv_endpoint $head_n:29500 \
/workspace/mingpt/main.py
```

GPU Profiling



Why profile your code?

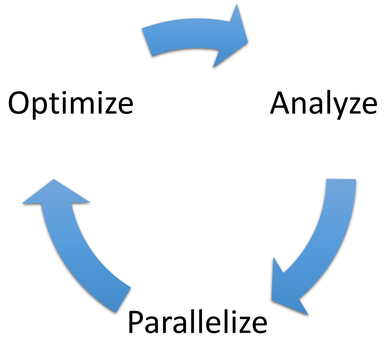
When is this useful?

- Scalable Deep Learning Application = functionally correct code + Optimized training & inference pipeline
- Code profiling gives you deeper insights into how your code performs at runtime.
- Allows you to focus your efforts on the problem areas of your application if optimizations are needed.
- Optimization does not always refer to execution time or latency, you can also optimize memory usage or throughput.
- Profiling can help you find errors or easy optimization opportunities.
- Often the way code executes on a device doesn't match what we would expect based on reading the code – profiling can help you figure out why.

When to profile your code?

Is it time to profile yet?

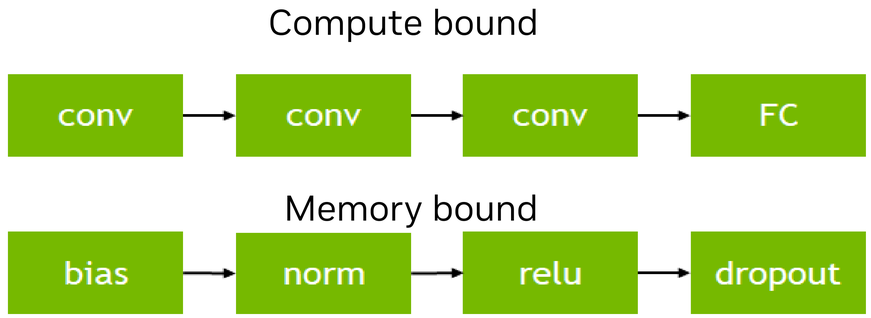
- Focus your efforts initially on getting a readable, maintainable, and correct version of your program finished.
  - Correctness beats everything else, nobody cares how fast or memory efficient your program is if it is unstable or gives the wrong result.
  - Example : "for loop" : limited unrolling because of less CPU Cores.
- Avoid premature optimization, bottlenecks can typically be attributed to a small fraction (<10%) of the overall code. Humans often aren't great at predicting which parts of the code that will be.
- Make sure your code is modular, break tasks into small easily readable functions with descriptive names.
- Equilibrium resource utilization. Ex. Performance Latency vs Memory utilization.
- Decide what to optimize, it may not be latency/execution time. It might be more important to optimize for memory usage or cost/throughput.
- Follow the Development Cycle:
  - Analyze
  - Parallelize
  - Optimize



Finding Performance Opportunities

Models can be data 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



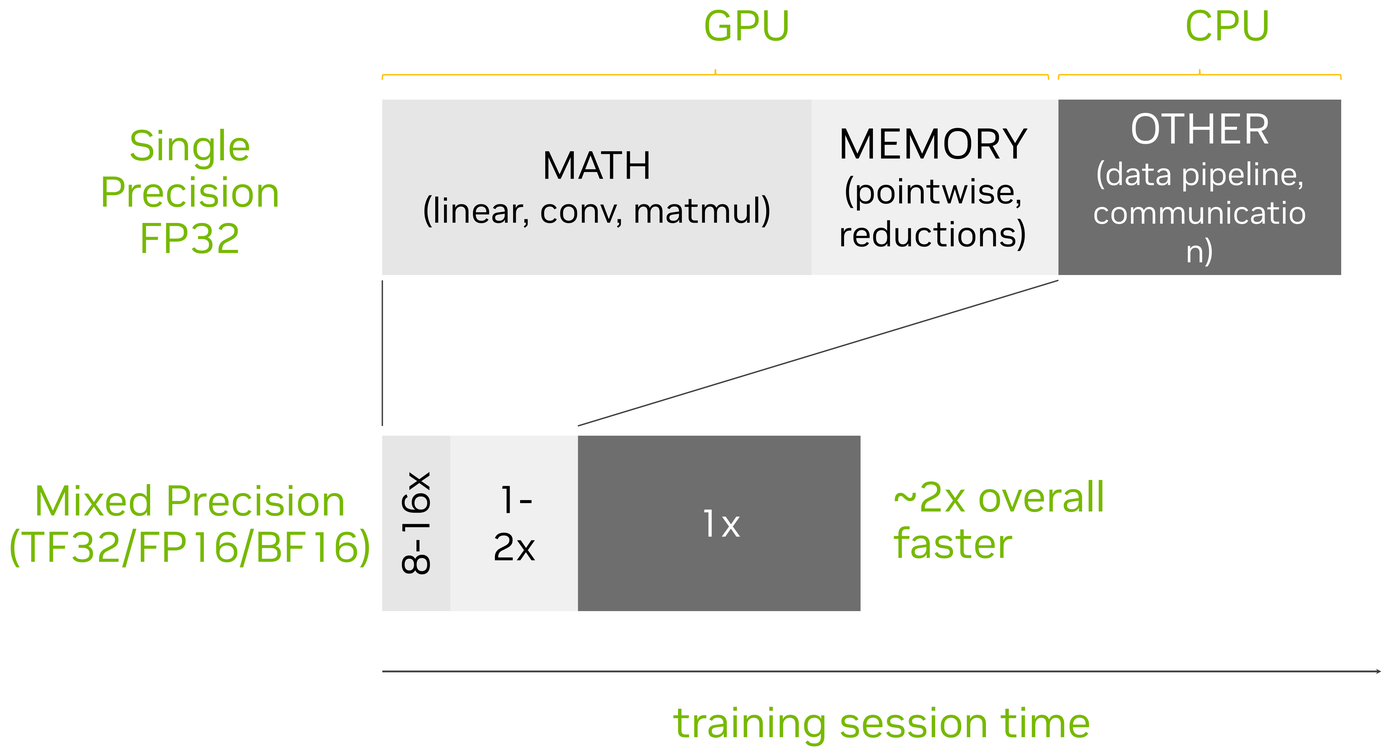
Compute heavy ops see speed-ups from GPUs

Limits of Performance Optimization

End-to-end perf depends on training composition

- **Amdahl's Law**:

If you speed up part of your training session (GPU work), then the remaining parts (CPU work) limit your overall performance



Deep Learning Optimization

Performance Analysis at System and DNN Level & Visualization

- Profiling will help in understanding of bottlenecks of Deep Learning applications at
  - System level : Data Transfer, Topology aware primitive, Kernels, Memory Architecture etc.
  - Neural Network level: Architecture Layers, inference batch, etc.

System Level Tuning

- System Tuning
  - Thread Synchronization, Multi GPU and node communication
  - Memory management & Kernel
    - profiling
- Leveraging/Optimizing Hardware
- Input Pipeline Optimization
- Many others….

DNN Level Tuning

- Training : Model Level tuning
- GPU Functions : host(CPU) <-> device(GPU)
- Inference: Latency & Throughput.
- Algorithm Techniques & Data Representations
- Pruning
- Calibration
- Quantization
- Many others….

DL Profiling Needs of 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

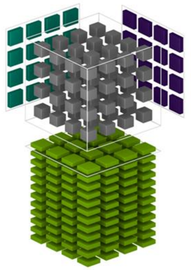
Sysadmins & DevOps

Optimized utilization and uptime, monitor GPU workloads, leverage hardware

Tools & Technologies

Profiling and monitoring tools for all users

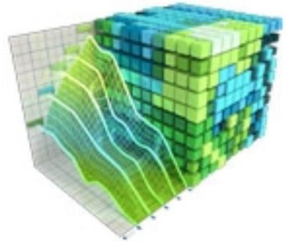
Researchers & Developers



NVTX
Nsight Systems
Nsight Compute

Skills in Algorithms

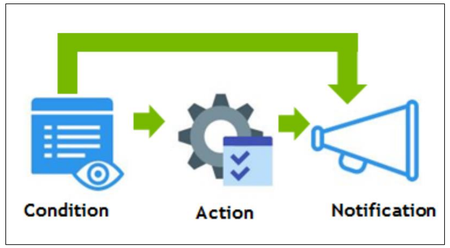
Data Scientists & Applied Researchers



Deep Learning Profiling (TF/PyTorch)
Nsight Systems with NVTX

Skills in Domains & Applications

Sysadmins & DevOps



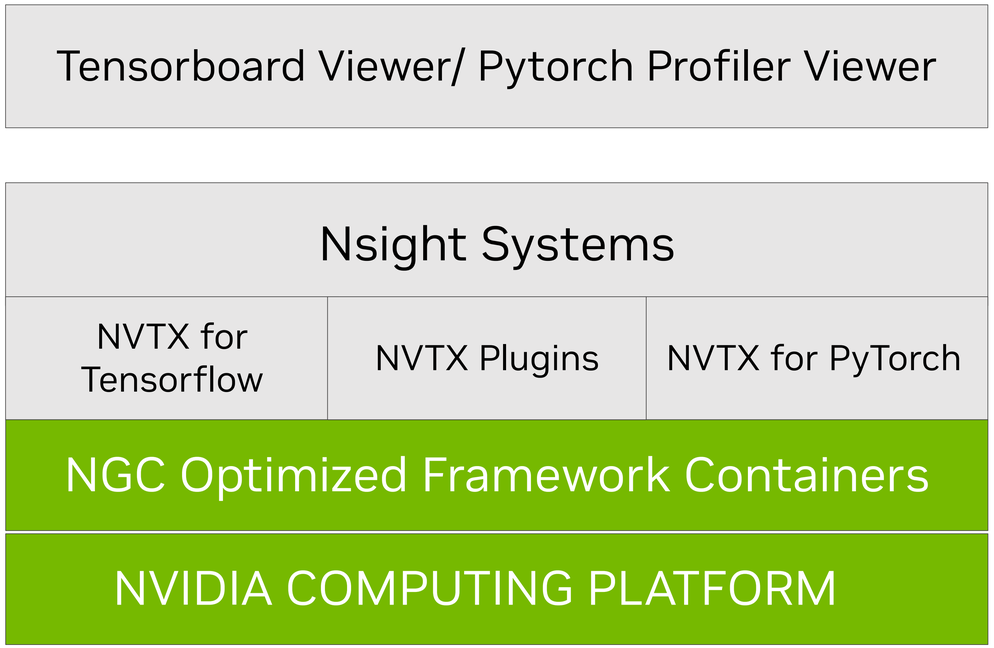
Data Center Monitoring Tools
DCGM, NVML

Skills in Systems

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



NSIGHT SUITE OF PROFILERS

NSIGHT Product Family

Nsight Systems

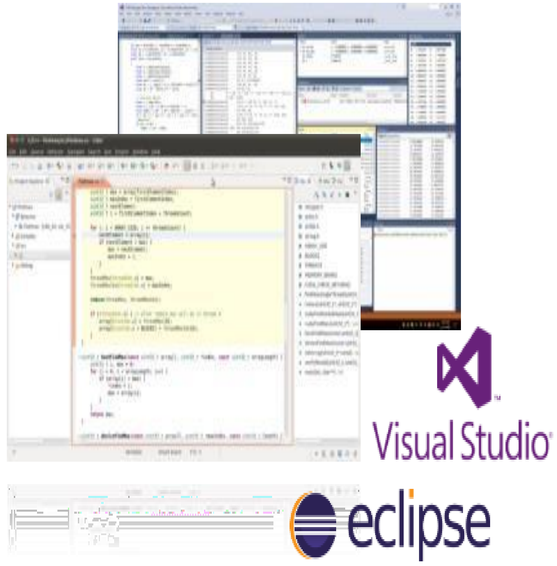
System-wide application algorithm tuning

Nsight Compute

CUDA API Debugging & Kernel Profiling

Nsight Graphics

Graphics Debugging & Profiling

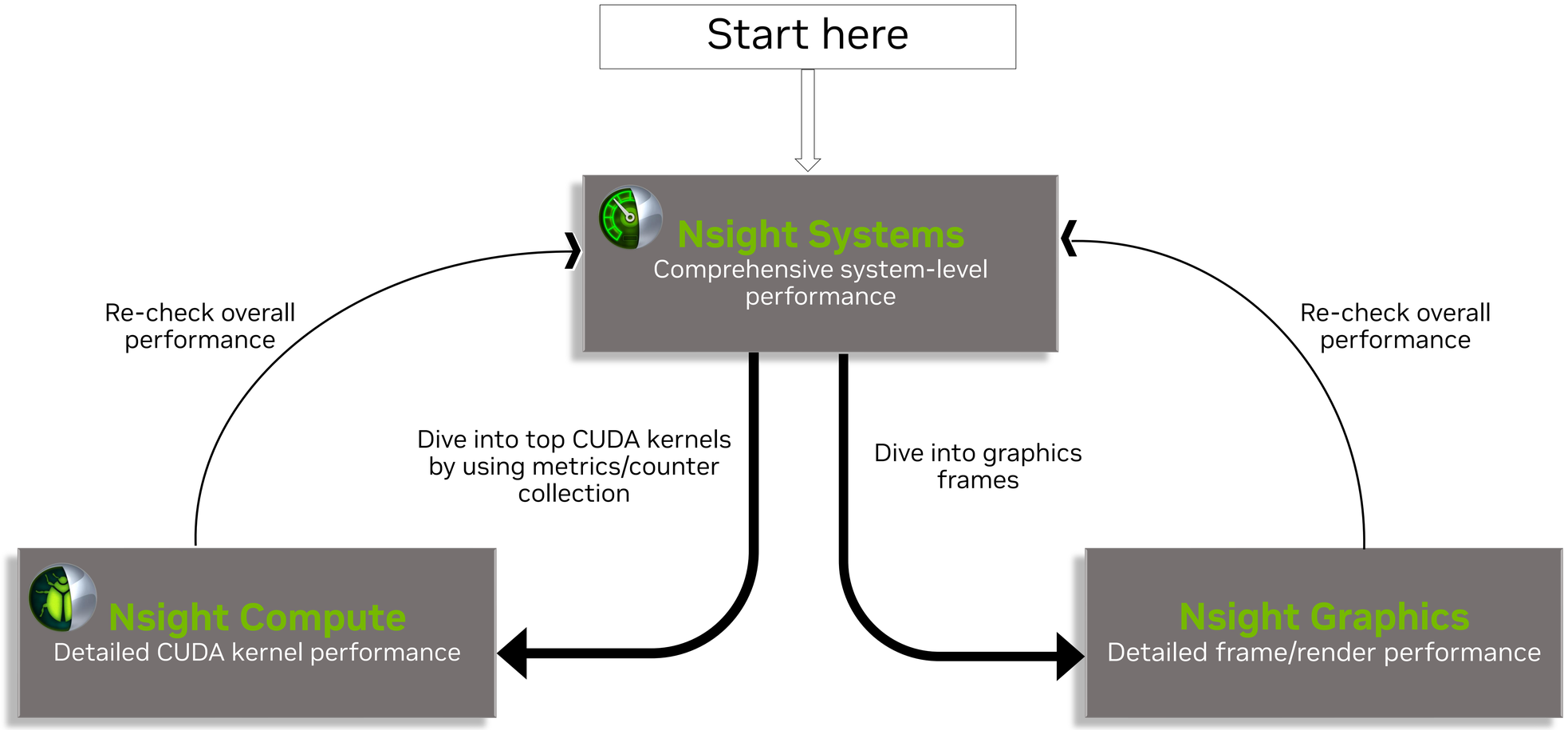


IDE Plugins

Nsight Eclipse Edition/Visual Studio (Editor, Debugger)

NSIGHT (Standalone) Tools Workflow

Nsight Systems, Nsight Compute and Nsight Graphics



NSIGHT SYSTEMS

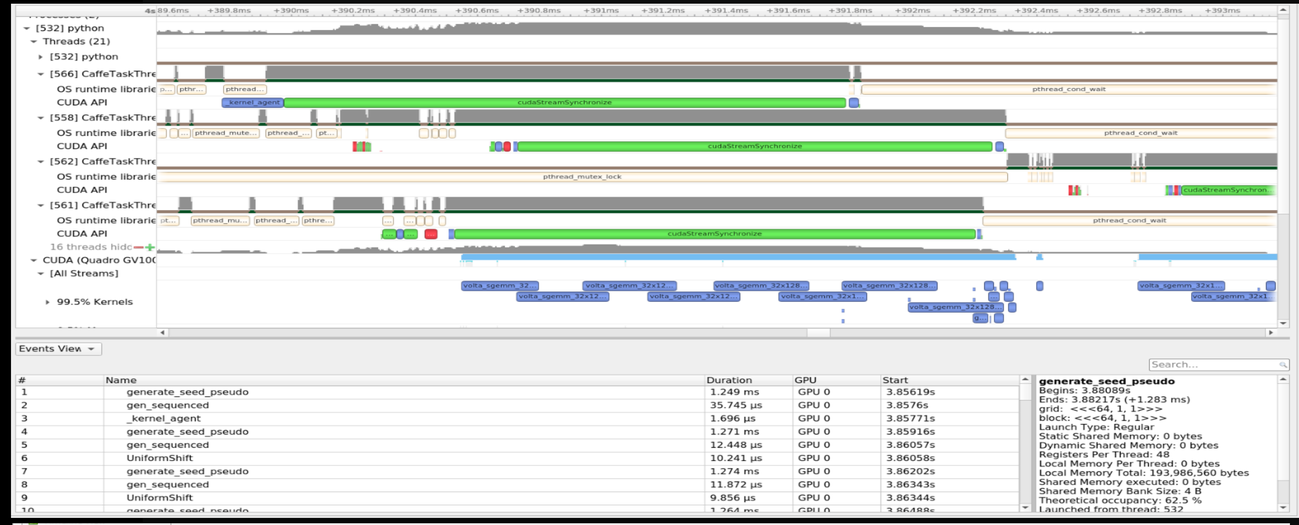
System Profiler

Key Features:

- System-wide application algorithm tuning
  - Multi-process tree support
- Locate optimization opportunities
  - Visualize millions of events on a very 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.
- Command Line, Standalone, IDE Integration

OS: Linux (x86, Power, Arm SBSA, Tegra), Windows, MacOSX (host)

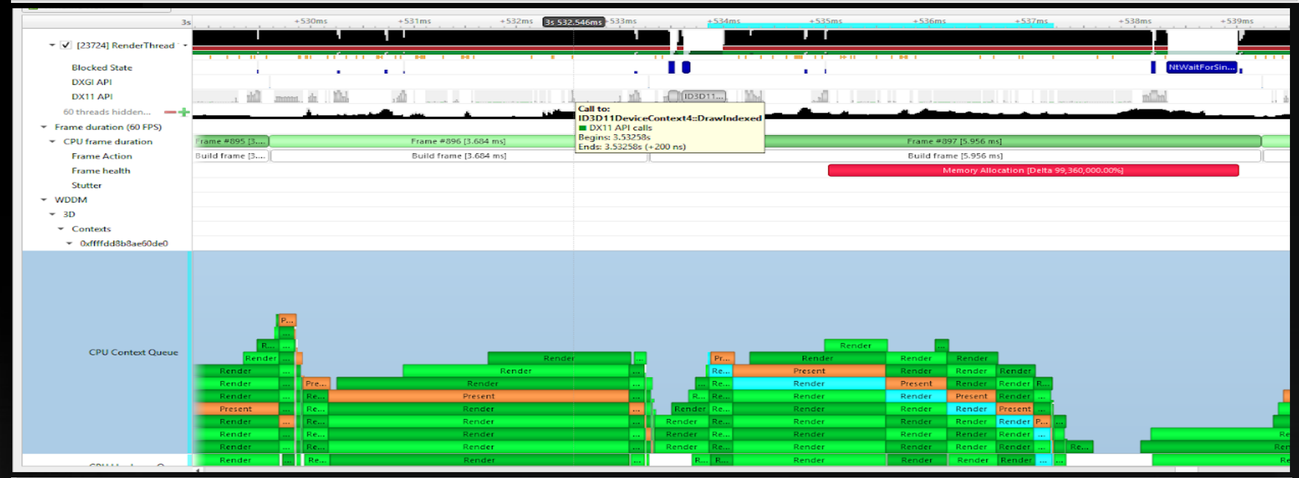
Documentation: https://developer.nvidia.com/nsight-systems



Events View

#	Name	Duration	GPU	Start
1	generate_seed_pseudo	1.249 ms	GPU 0	3.85619s
2	gen_sequenced	35.745 µs	GPU 0	3.8576s
3	_kernel_agent	1.696 µs	GPU 0	3.85771s
4	generate_seed_pseudo	1.271 ms	GPU 0	3.85916s
5	gen_sequenced	12.448 µs	GPU 0	3.86057s
6	UniformShift	10.241 µs	GPU 0	3.86058s
7	generate_seed_pseudo	1.274 ms	GPU 0	3.86202s
8	gen_sequenced	11.872 µs	GPU 0	3.86343s
9	UniformShift	9.856 µs	GPU 0	3.86344s

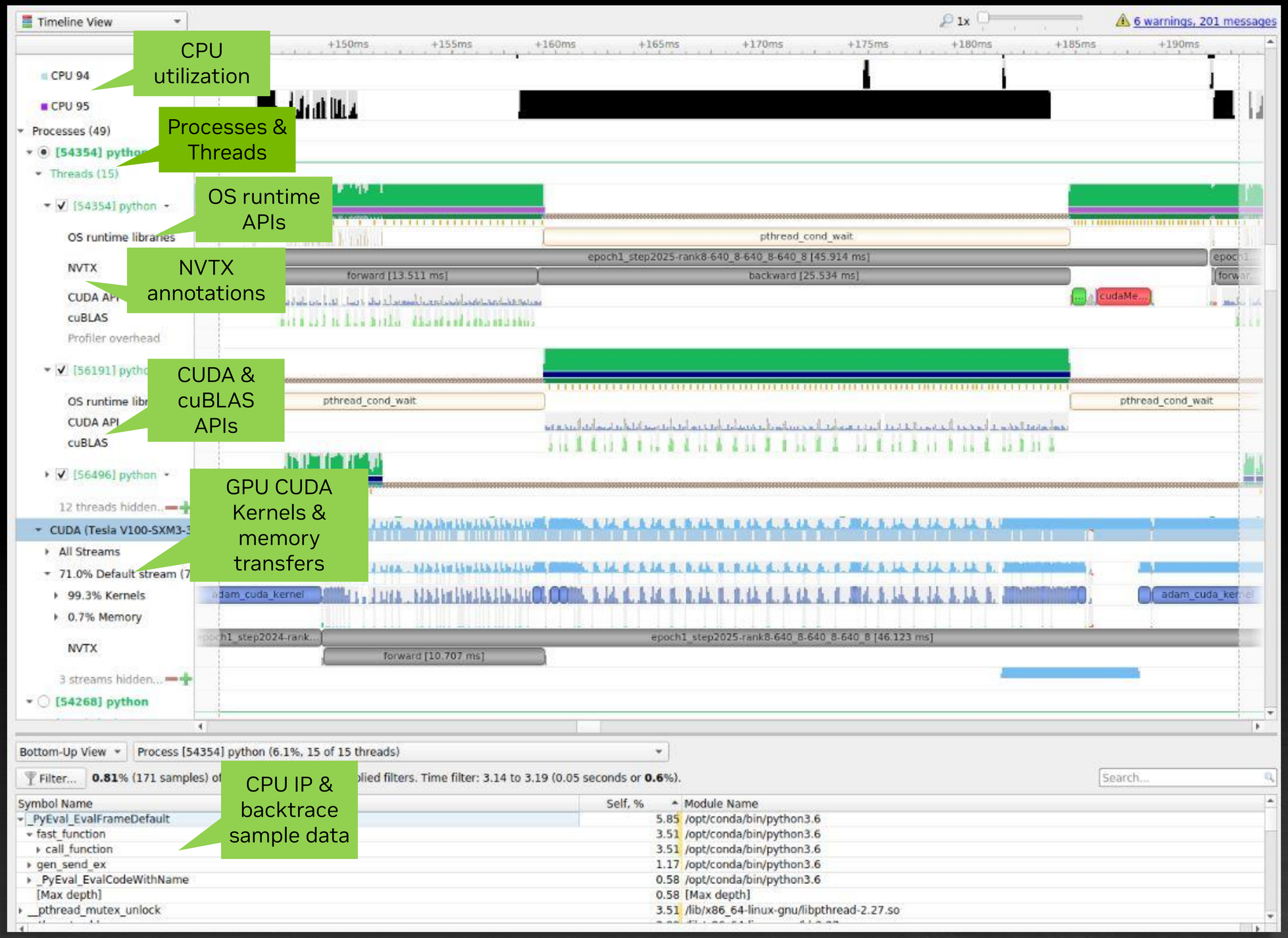
generate_seed_pseudo
Begins: 3.88089s
Ends: 3.88217s (+1.283 ms)
grid: <<<64, 1, 1>>>
block: <<<64, 1, 1>>>
Launch Type: Regular
Static Shared Memory: 0 bytes
Dynamic Shared Memory: 0 bytes
Registers Per Thread: 48
Local Memory Per Thread: 0 bytes
Local Memory Total: 193,986,560 bytes
Shared Memory executed: 0 bytes
Shared Memory Bank Size: 4 B
Theoretical occupancy: 62.5 %
Launched from thread: 532



Bottom-Up View | Process [9695] vmd_LINUXAMD64.11 (3 of 19 threads)

Filter... **99.82%** (23,260 samples) of data is shown due to applied filters.

Symbol Name	Self, %	Module Name
VolumetricData::compute_volume_gradient()	20.14	/home/johns/vmd/src/gtcbuilds/vmd_LINUXAMD64.11
VolumetricData::compute_volume_gradient()	20.14	/home/johns/vmd/src/gtcbuilds/vmd_LINUXAMD64.11
BaseMolecule::add_volume_data(char const*, double const*, double const*, double const*, double const*, int, int, int, float*)	18.30	/home/johns/vmd/src/gtcbuilds/vmd_LINUXAMD64.11
VMDApp::molecule_add_volumetric(int, char const*, double const*, double const*, double const*, double const*, int, int, int, float*)	18.30	/home/johns/vmd/src/gtcbuilds/vmd_LINUXAMD64.11
obj_segmentation(void*, Tcl_Interp*, int, Tcl_Obj* const*)	18.30	/home/johns/vmd/src/gtcbuilds/vmd_LINUXAMD64.11
[Max depth]	18.30	[Max depth]
BaseMolecule::add_volume_data(char const*, float const*, float const*, float const*, float const*, int, int, int, float*, float*, float*)	1.84	/home/johns/vmd/src/gtcbuilds/vmd_LINUXAMD64.11
MolFilePlugin::read_volumetric(Molecule*, int, int const*)	1.84	/home/johns/vmd/src/gtcbuilds/vmd_LINUXAMD64.11
VMDApp::molecule_load(int, char const*, char const*, FileSpec const*)	1.84	/home/johns/vmd/src/gtcbuilds/vmd_LINUXAMD64.11
text_cmd_mol(void*, Tcl_Interp*, int, char const**)	1.84	/home/johns/vmd/src/gtcbuilds/vmd_LINUXAMD64.11
TclInvokeStringCommand	1.84	/home/johns/vmd/src/gtcbuilds/vmd_LINUXAMD64.11
TclEvalObjvInternal	1.84	/home/johns/vmd/src/gtcbuilds/vmd_LINUXAMD64.11
TclExecuteByteCode	1.84	/home/johns/vmd/src/gtcbuilds/vmd_LINUXAMD64.11
TclCompEvalObj	1.84	/home/johns/vmd/src/gtcbuilds/vmd_LINUXAMD64.11
TclEvalObjEx	1.84	/home/johns/vmd/src/gtcbuilds/vmd_LINUXAMD64.11
Tcl_RecordAndEvalObj	1.84	/home/johns/vmd/src/gtcbuilds/vmd_LINUXAMD64.11
TclTextInterp::evalFile(char const*)	1.84	/home/johns/vmd/src/gtcbuilds/vmd_LINUXAMD64.11
VMDApp::logfile_read(char const*)	1.84	/home/johns/vmd/src/gtcbuilds/vmd_LINUXAMD64.11
VMDreadStartup(VMDApp*)	1.84	/home/johns/vmd/src/gtcbuilds/vmd_LINUXAMD64.11
[Max depth]	1.84	[Max depth]
0x7f10ca7022d6	5.13	/usr/lib64/libcuda.so.390.25
obj_segmentation(void*, Tcl_Interp*, int, Tcl_Obj* const*)	3.44	/home/johns/vmd/src/gtcbuilds/vmd_LINUXAMD64.11



Additionally:

Trace:

- TensorRT
- Direct3D11,12,DXR
- Vulkan
- OpenGL
- OpenACC
- MPI
- OpenMP
- Ftrace
- ETW
- WDDM
- GPU Context Switch

Export:

- SQLite
- HDF5
- JSON

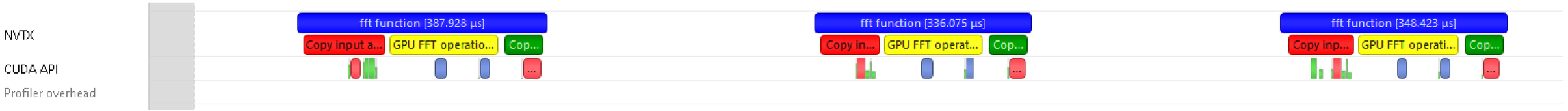
Architectures:

- X86_64
- Power
- Arm SBSA
- Tegra

NVTX: NVIDIA Tools Extension API

- Code Annotation tool to mark functions or chunks of code.
- NVTX and Nsight Systems together are powerful tools for visualizing CPU and GPU performance
- Workflow:
  - Use decorators *@nvtx.annotate()* or context manager *with nvtx.annotate(..):* to mark code to be measured.
  - Run it with Nsight Systems.
    - *$ nsys profile -t nvtx,osrt --force-overwrite=true --stats=true --output=quickstart python nvtx-quickstart.py*
    - *$ nsys –help or nsys [specific command] --help*
  - nsys_demo.nsys-rep and sqlite database generated to be viewed on Nsight Systems UI.

```python
import numpy as np
import cupy as cp
import nvtx

@nvtx.annotate("fft function", color="blue")
def fast_fft(input_array):
    with nvtx.annotate("Copy input array to GPU and CuPy", color="red"):
        gpu_array = cp.array(input_array)
    with nvtx.annotate("GPU FFT operation", color="yellow"):
        result = cp.fft.fft(gpu_array)
    with nvtx.annotate("Copy back to CPU and Numpy", color="green"):
        cpu_result = result.get()
    return cpu_result


for _ in range(5):
    print(fast_fft(np.random.random(10)))
```



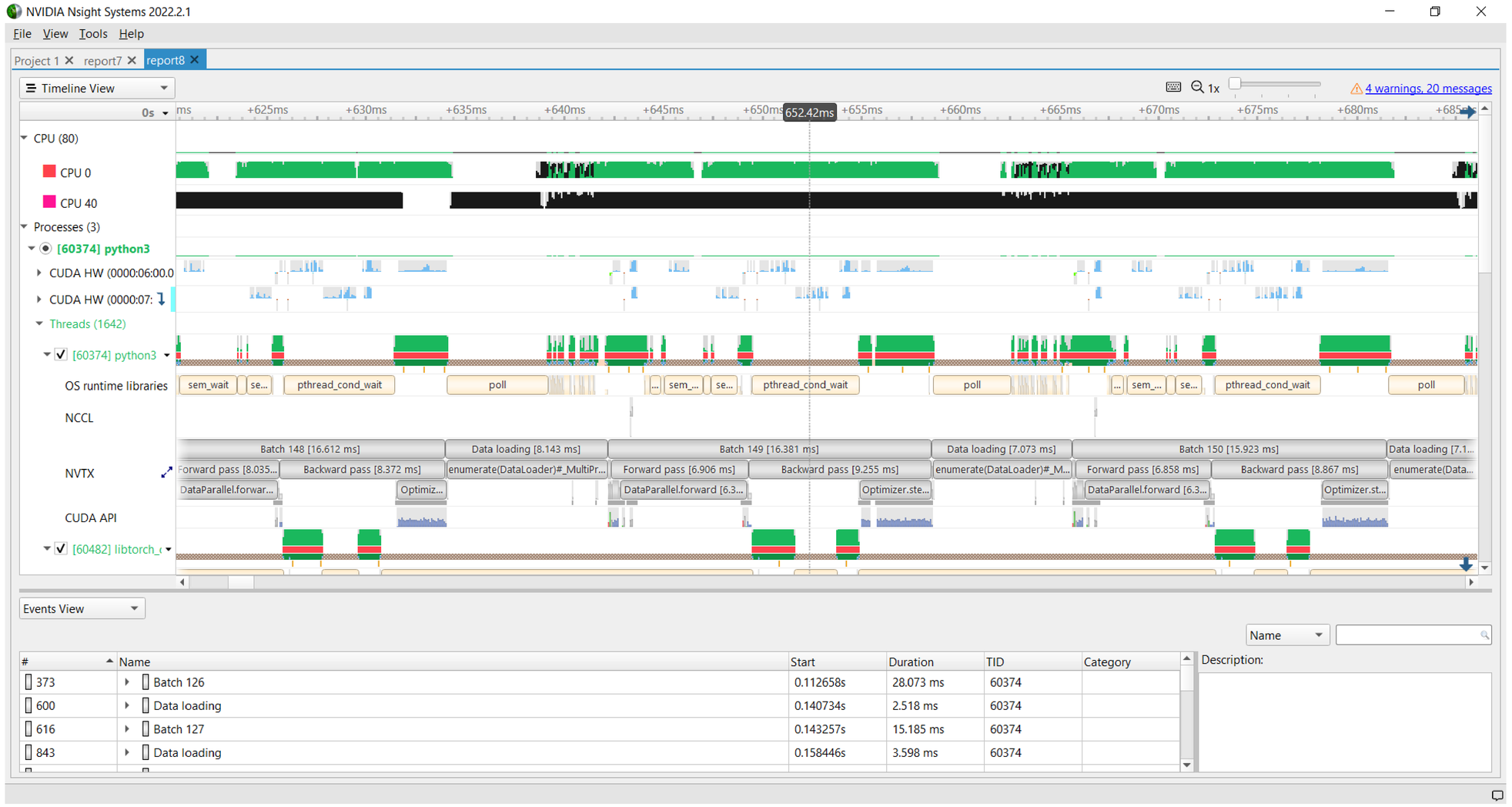
Results in Nsight Systems UI - report generated with `"nsys profile –o nvtx_test python test.py"`

NSIGHT Systems

Key Features

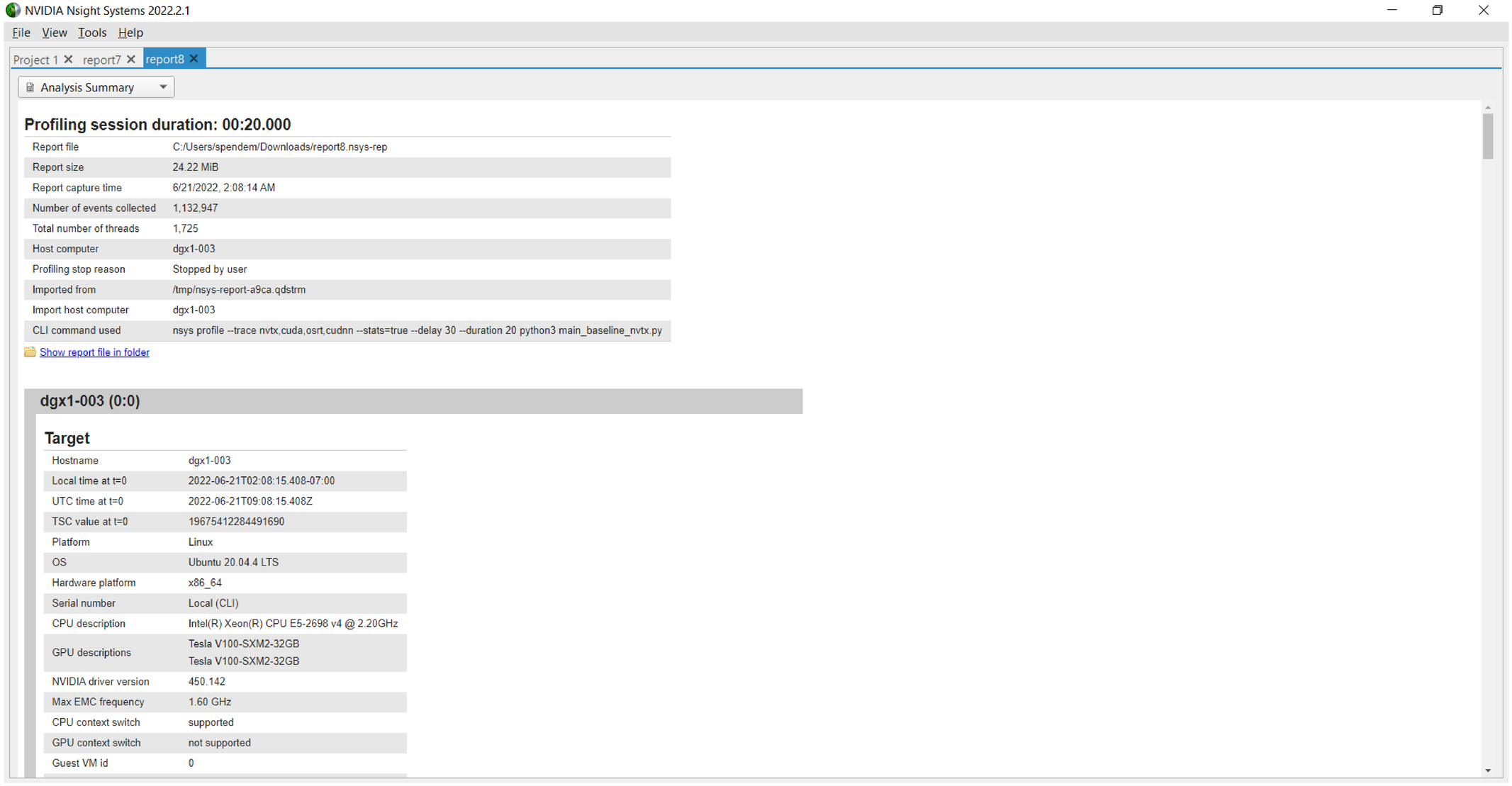
Timeline View →

Event Table →



NSIGHT Systems

Analysis Summary



NSIGHT Systems

Diagnostics Summary

NVIDIA Nsight Systems 2022.2.1

File View Tools Help

Project 1 ✕ report7 ✕ report8 ✕

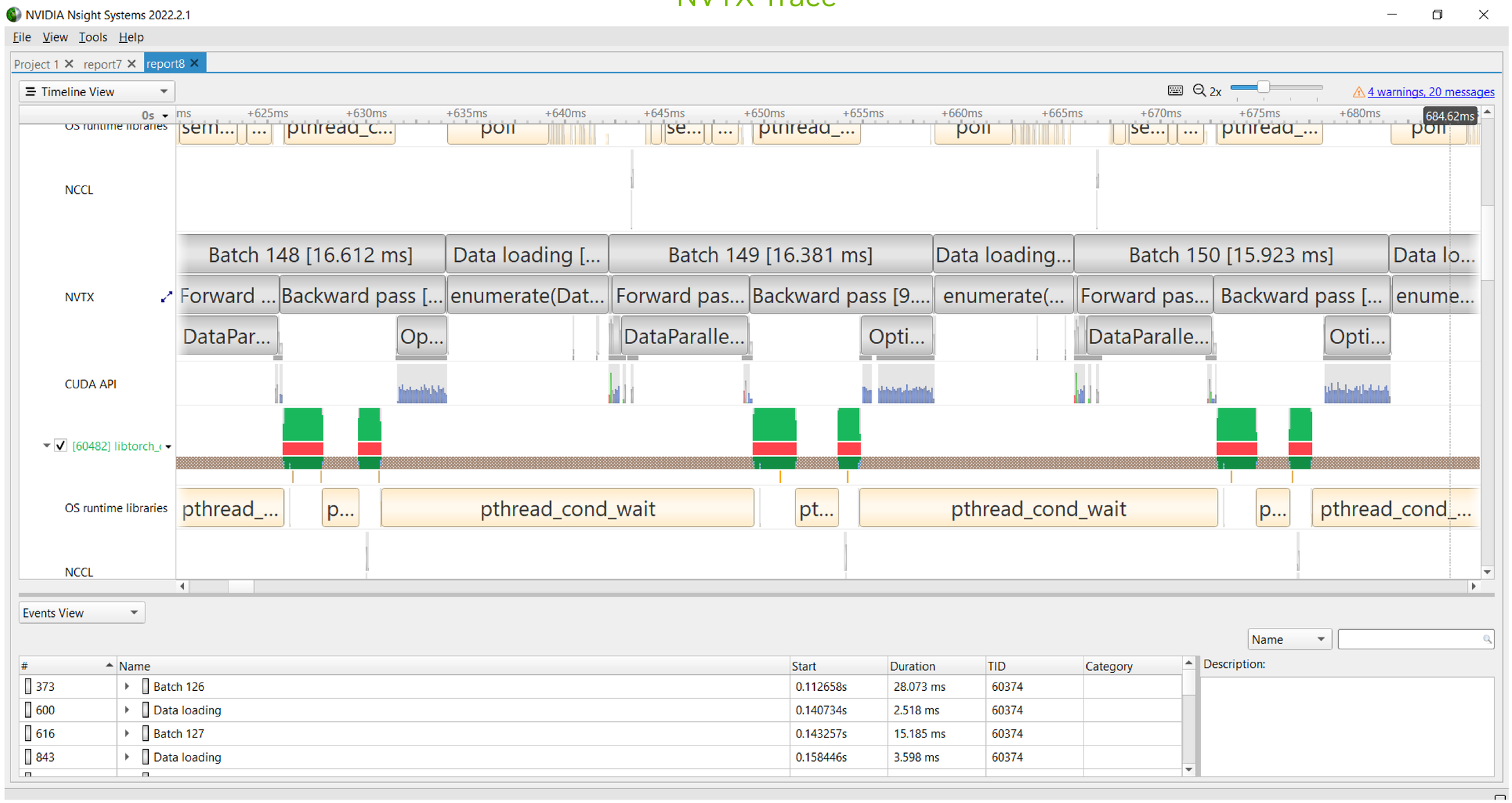
Diagnostics Summary

Messages

	Source	Process ID	Time	Description
ℹ	Daemon	60374	-00:30.702	Process was launched by the profiler, see /tmp/nvidia/nsight_systems/quadd_session_1060356/streams/pid_60374_stdout.log and stderr.log for program output
ℹ	Injection	60374	-00:30.612	Common injection library initialized successfully.
ℹ	Injection	60374	-00:30.597	OS runtime libraries injection initialized successfully.
ℹ	Injection	60374	-00:30.358	cuDNN injection initialized successfully.
ℹ	Injection	60374	-00:30.159	OpenMP injection initialized successfully.
ℹ	Injection	60374	-00:29.719	Buffers holding CUDA trace data will be flushed on CudaProfilerStop() call.
ℹ	Injection	60374	-00:29.451	CUDA injection initialized successfully.
ℹ	Injection	60374	-00:28.117	NVTX injection initialized successfully.
ℹ	Daemon		-00:00.188	Intel(c) Last Branch Record (LBR) backtraces collected.
ℹ	Daemon		-00:00.188	Hardware event 'instructions', with sampling period 1800000, used to trigger sample collection.
ℹ	Daemon		-00:00.000	1 CPU IP samples collected for every CPU IP backtrace collected.
ℹ	Analysis		00:00.000	Profiling has started.
⚠	Analysis	60374	00:04.280	Not all NVTX events might have been collected.
ℹ	Analysis	60374	00:04.280	Number of NVTX events collected: 434,574.
⚠	Analysis	60374	00:04.280	Not all CUDA events might have been collected.
ℹ	Analysis	60374	00:04.280	Number of CUDA events collected: 374,419.
⚠	Analysis	60374	00:04.280	Not all OS runtime libraries events might have been collected.
ℹ	Analysis	60374	00:04.280	Number of OS runtime libraries events collected: 50,873.
⚠	Analysis	60374	00:04.280	Not all cuDNN events might have been collected.
ℹ	Analysis	60374	00:04.280	Number of cuDNN events collected: 8,140.
ℹ	Analysis		00:20.000	Profiling has stopped.
ℹ	Injection	60374	00:20.067	Number of CUPTI events produced: 397,345, CUPTI buffers: 50.
ℹ	Injection	60374	00:20.067	The cudaProfilerStart API was ignored 1 times due to configuration settings; Note that, when requested, only the first pair of cudaProfilerStart/Stop APIs, after the collection is started, will be effective.
ℹ	Daemon		00:20.901	Number of IP samples collected: 30,878.

NSIGHT Systems

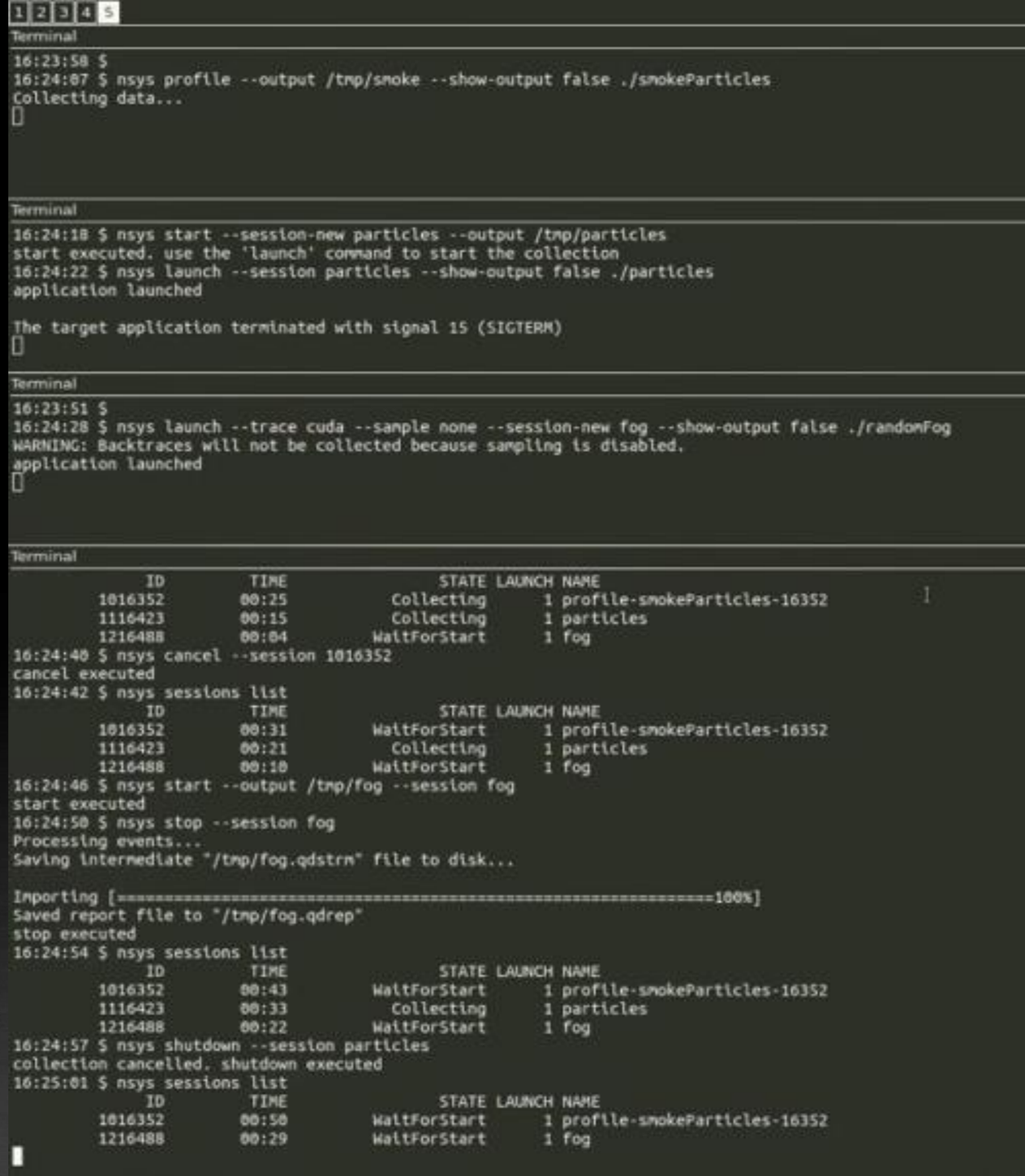
NVTX Trace



NSIGHT SYSTEMS

Other Features

- CLI improvement
  - Simultaneous sessions & management
  - nsys launch forwards terminal and signals
  - nsys stats command
  - nsys export command
- CUDA context “All Streams” aggregation tree
- Windows WDDM GPU memory usage, paging, evictions
- Graphics trace enhancements



NSIGHT COMPUTE

NSIGHT COMPUTE

Kernel Profiling Tool

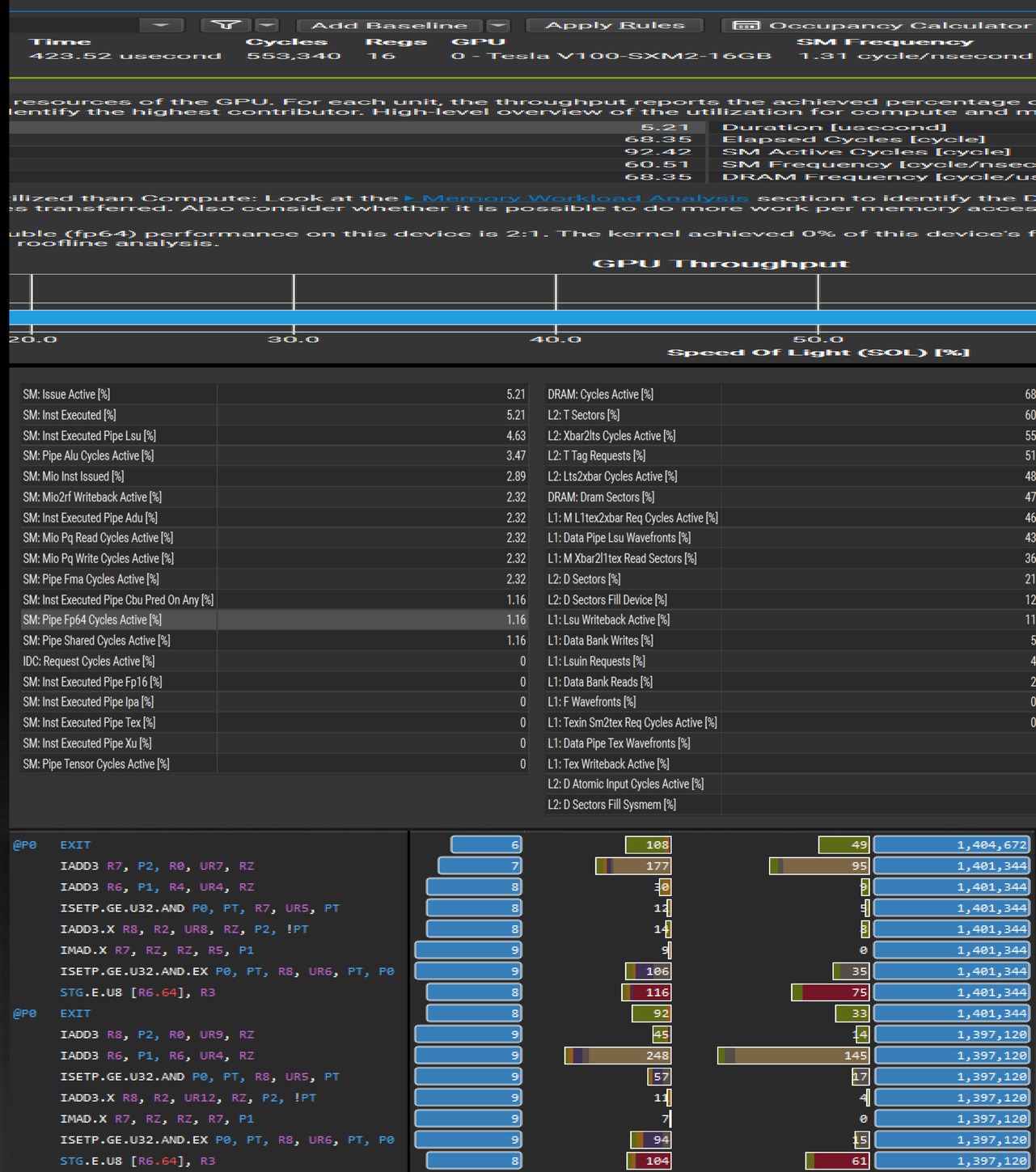
Key Features:

- Interactive CUDA API debugging and kernel profiling
- Fast Data Collection
- Compare performance metrics across different runs
  - *$ ncu --set full –k <kernel_name> -s 4 –c 1 –o myreport ./myapp*

OS: Linux (x86, Power, Tegra, Arm SBSA), Windows, MacOSX (host only)

GPUs: Volta, Turing, A100 GPUs

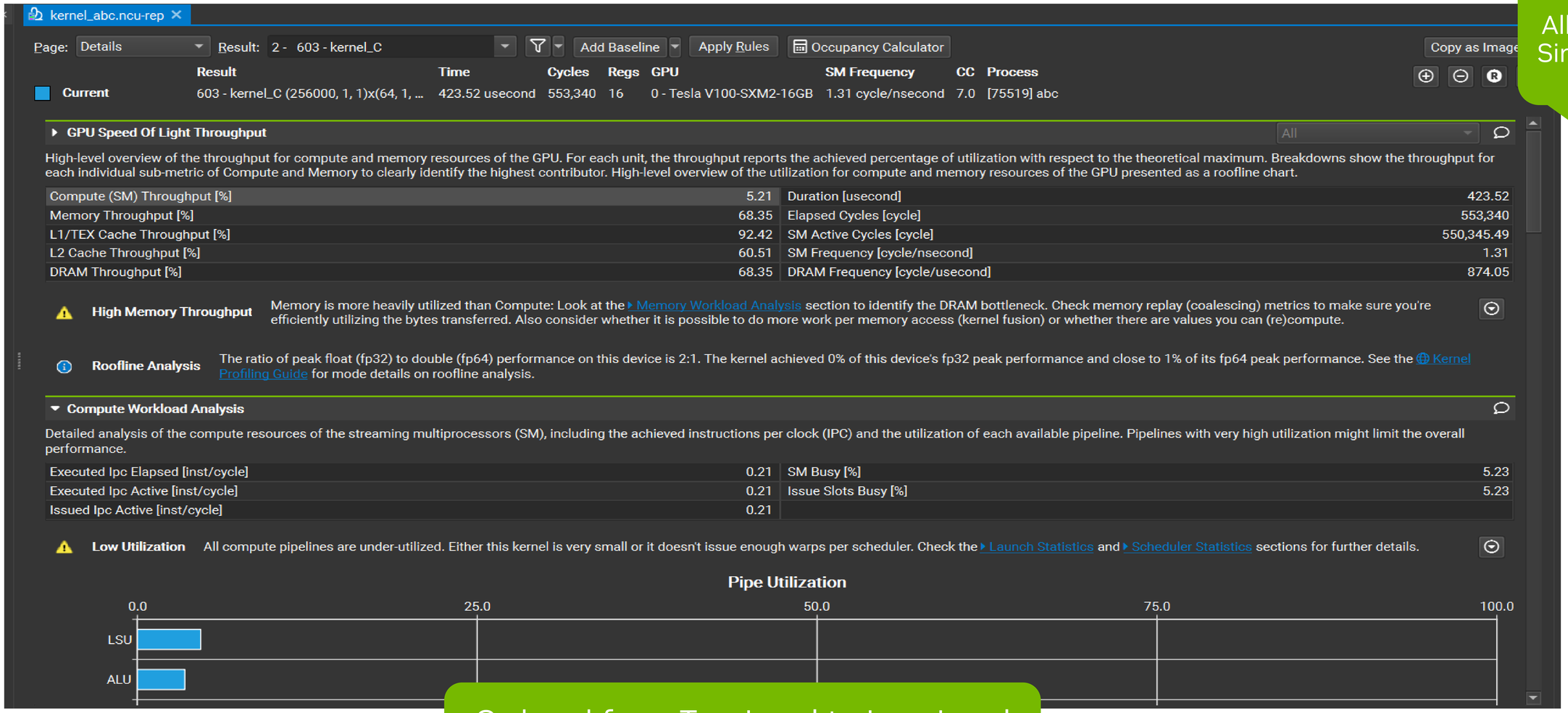
Documentation: https://developer.nvidia.com/nsight-compute



NSIGHT Compute

Profile Report – Details Page

All Data on Single Page

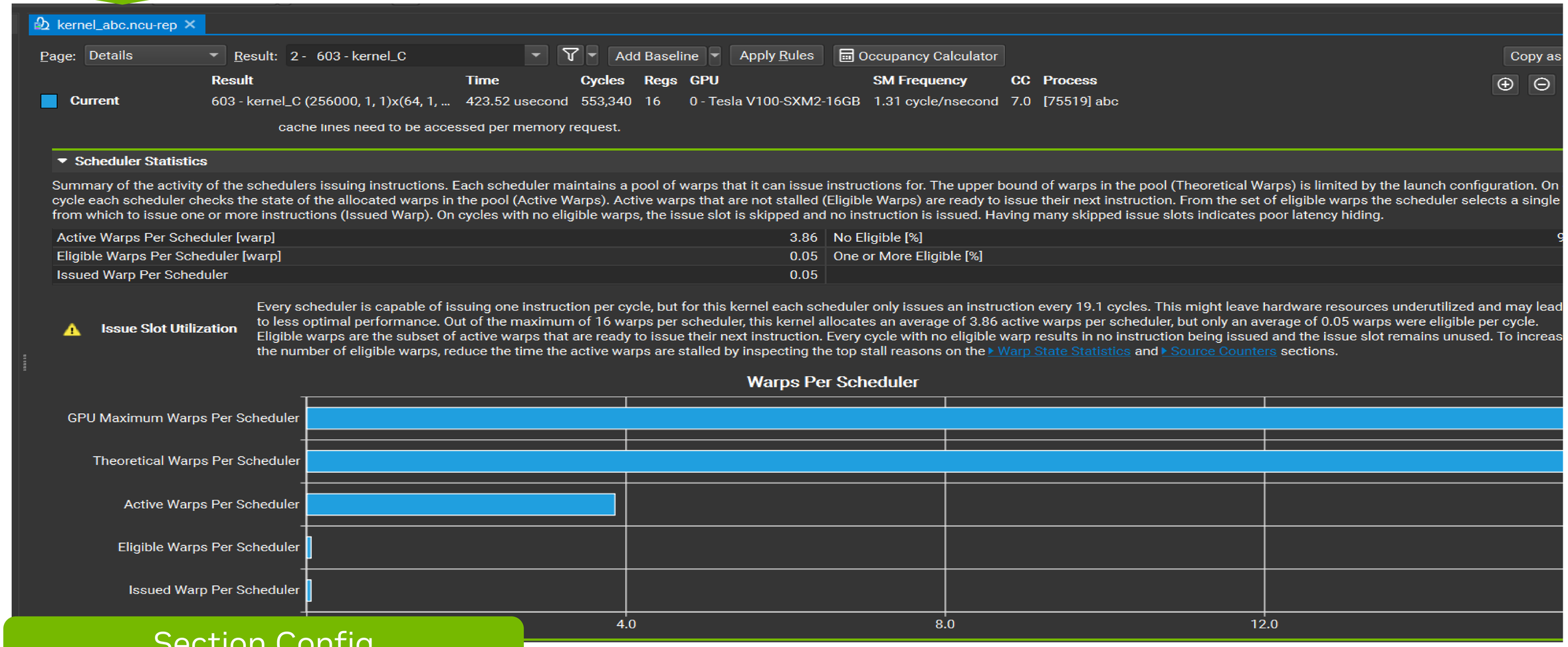


Ordered from Top-Level to Low-Level

NSIGHT Compute

Section Example

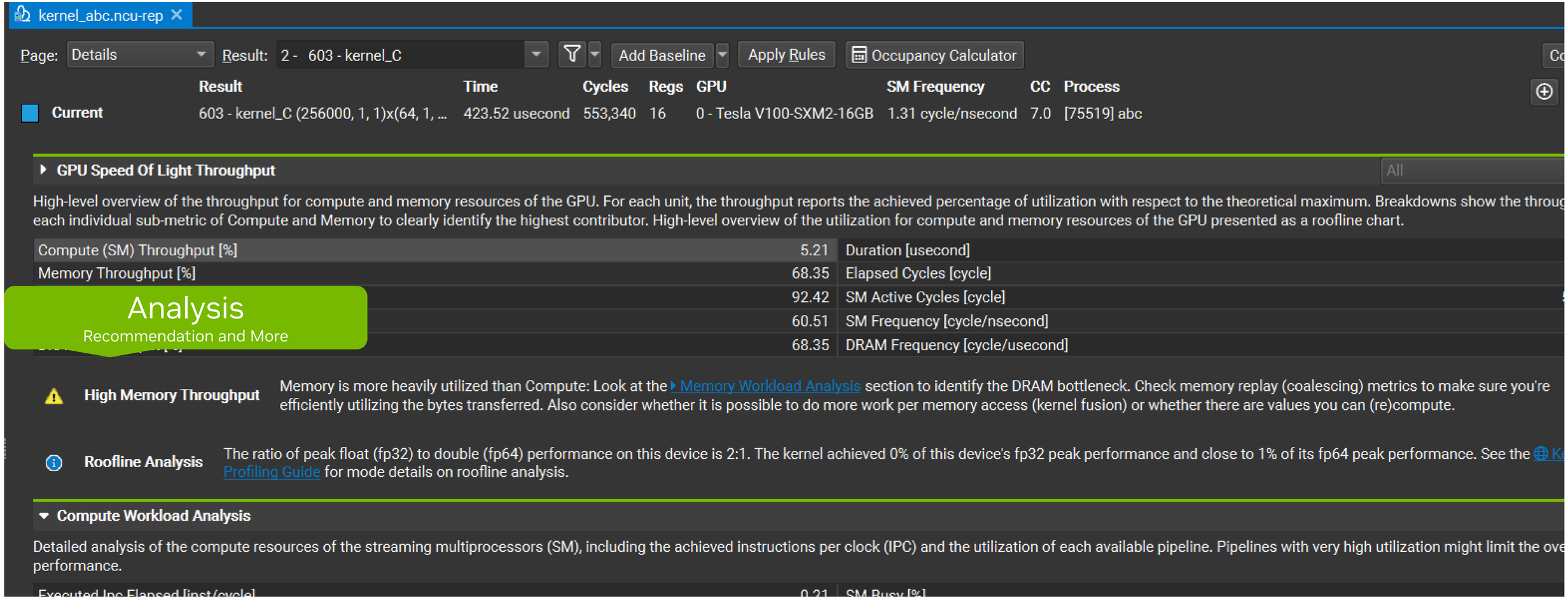
Section Header
provides overview & context for other sections



Section Config
completely data driven add/modify/change sections

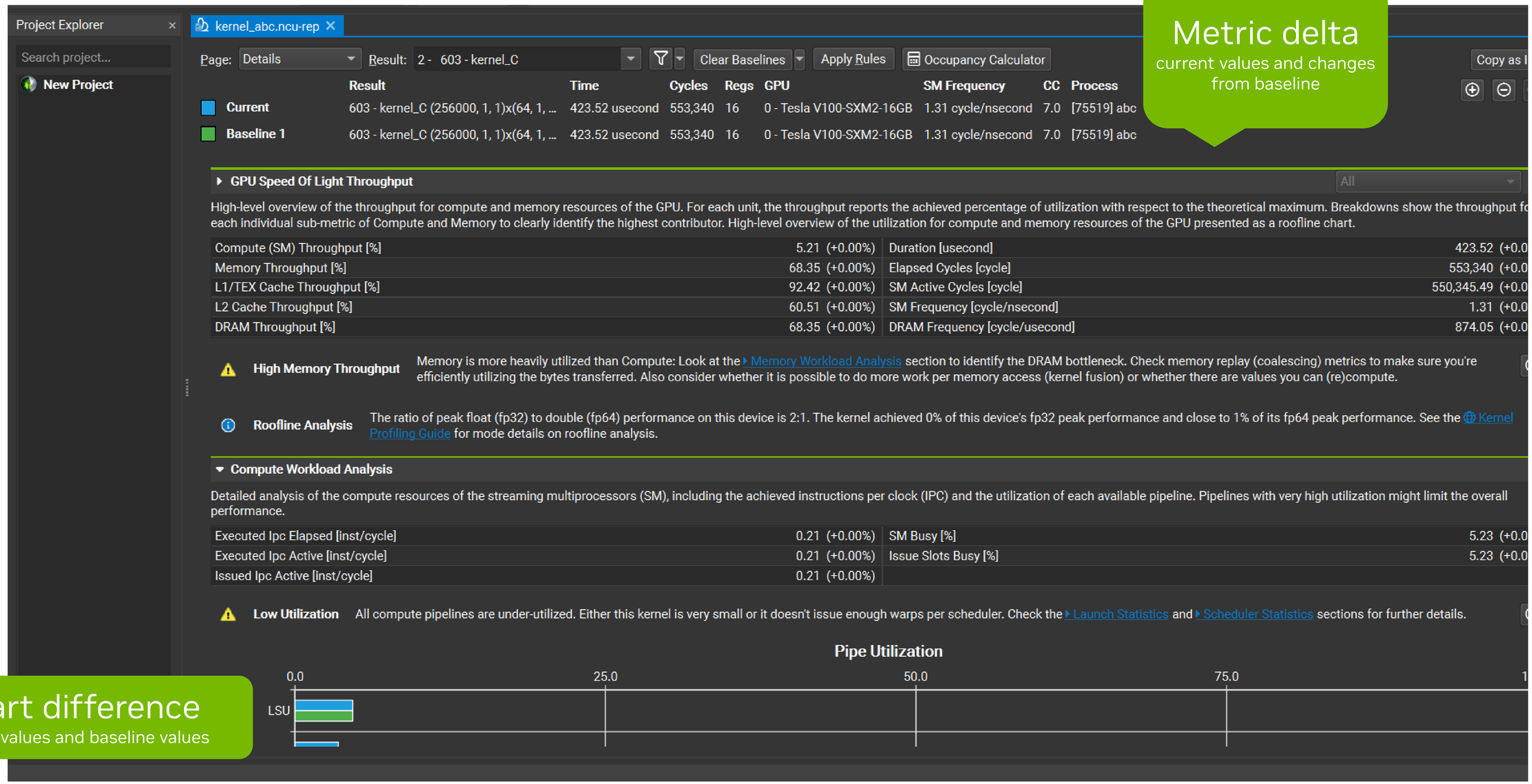
NSIGHT Compute

Unguided Analysis / Rules System



NSIGHT Compute

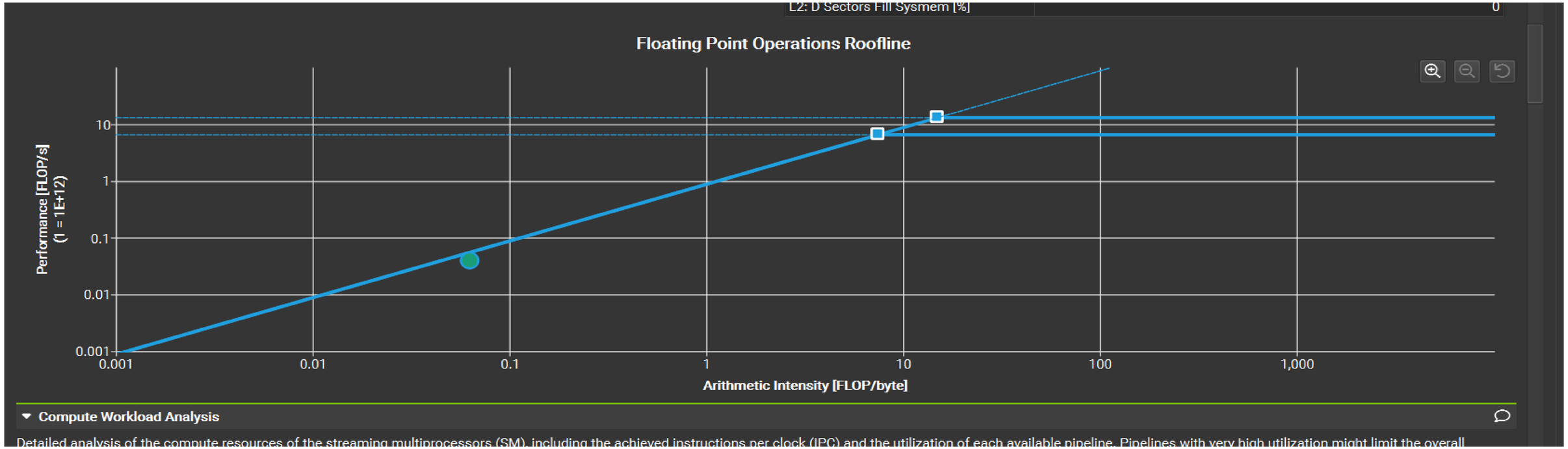
Differencing kernel runs



NSIGHT Compute

New Roofline Analysis

- Efficient way to evaluate kernel characteristics, quickly understand potential directions for further improvements or existing limiters
- Inputs: Arithmetic Intensity (FLOPS/bytes)
  Performance (FLOPS/s)
- Ceilings: Peak Memory Bandwidth
  Peak FP32/FP64 Performance



TENSORFLOW profiler

Tensorboard Profile

TensorBoard SCALARS GRAPHS DISTRIBUTIONS HISTOGRAMS PROFILE INACTIVE

CAPTURE PROFILE

Runs (2)

20200314-180936/train/2020_03_...

Tools (5)

overview_page

Hosts (1)

72cefc0dc85c

Performance Summary

Average Step Time *lower is better* *(σ = 4.6 ms)* — 30.1 ms

- **All Others Time** *(σ = 1.3 ms)* — 3.2 ms
- **Compilation Time** *(σ = 0.0 ms)* — 0.0 ms
- **Output Time** *(σ = 0.0 ms)* — 0.0 ms
- **Input Time** *(σ = 4.4 ms)* — 25.2 ms
- **Kernel Launch Time** *(σ = 0.4 ms)* — 1.1 ms
- **Host Compute Time** *(σ = 0.1 ms)* — 0.2 ms
- **Device to Device Time** *(σ = 0.0 ms)* — 0.0 ms
- **Device Compute Time** *(σ = 0.0 ms)* — 0.4 ms

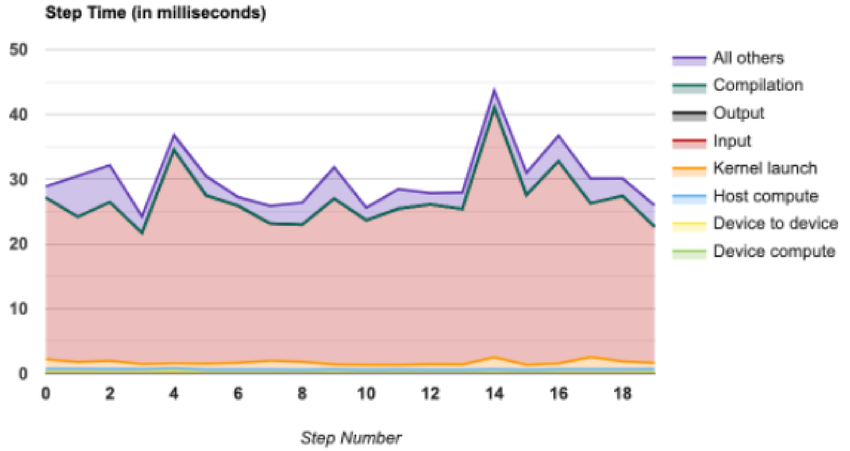
Device Compute Precisions *(out of Total Device Time)*

- 16-bit: 0.0
- 32-bit: 100.0

Run Environment

Number of Hosts used: 1

Step-time Graph



Recommendation for Next Step

- Your program is HIGHLY input-bound because 83.9% of the total step time sampled is waiting for input. Therefore, you should first focus on reducing the input time.
- 3.6 % of the total step time sampled is spent on Kernel Launch.
- 10.5 % of the total step time sampled is spent on All Others time.
- Only 0.0% of device computation is 16 bit. So you might want to replace more 32-bit Ops by 16-bit Ops to improve performance (if the reduced accuracy is acceptable).

Tool troubleshooting / FAQ

- Refer to the TF2 Profiler FAQ

https://www.tensorflow.org/tensorboard/tensorboard_profiling_keras

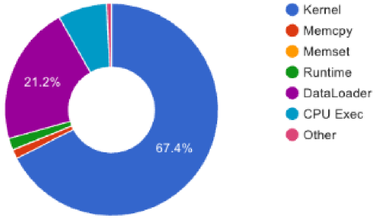
PYTORCH Profiler

Configuration

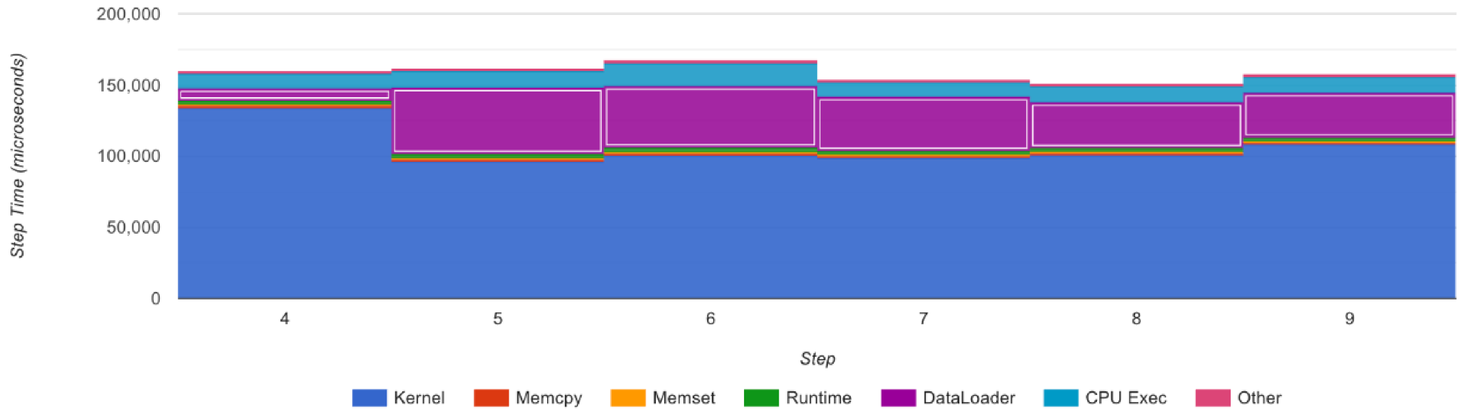
Number of Worker(s)	**1**
Device Type	**GPU**
Number of Device(s)	**1**

Execution Summary

Category	Time Duration (us)	Percentage (%)
Average Step Time	157,532	100
Kernel	106,185	67.41
Memcpy	2,213	1.4
Memset	65	0.04
Runtime	2,816	1.79
DataLoader	33,372	21.18
CPU Exec	11,720	7.44
Other	1,159	0.74



Step Time Breakdown



Performance Recommendation

- This run has high time cost on input data loading. 21.2% of the step time is in DataLoader. You could try to set num_workers on DataLoader's construction and enable multi-processes on data loading. Reference: Single- and Multi-process Data Loading

https://pytorch.org/blog/introducing-pytorch-profiler-the-new-and-improved-performance-tool/

Profiling Best Practices

Tips and Techniques

- Most of the Acceleration happens in the earlier stages of workload deployment - Data Loading/Copying etc.
- There should be as less as possible data movement/copying from GPU-CPU for efficiency. Use Batching.
- Always focus on the biggest time-consuming block first.
- For DL profiling, Run profiling only for 10-20 mini-batches (iterations)
  - DL training Operations are identical for every iteration; hence a few are sufficient
  - Always Use warmup steps before running Profiling
- Ensure AMP is enabled if your Tensor Core usage is zero.
- Examples of issues in the DL code and their fixes:
  - Change Interpolation to Deconvolution layer -> Deconvolutions have tensor core support and are generally faster than interpolations
  - Use 3x3 kernel instead of 7x7 -> 7x7 kernels require much more compute than 3x3 (Fast)
  - Use 32 filters instead of 30 for convolutions -> Tensor Cores achieve the best performance on layers which have the number of channels a multiple of 8.
- Rules for Efficient/Optimal Code:
  - No IO, (File reading/writing) inside loops
  - No loops – Parallelize
  - Lazy evaluation and iterators are incredibly useful
  - Skip execution or stop early whenever possible

Resources

Helpful Links for Reference. Learn More!

- Web: https://developer.nvidia.com/tools-overview
- NVTX Documentation: https://docs.nvidia.com/nsight-visual-studio-edition/2020.1/nvtx/index.html#nvtxt-file-extension
- DL performance Checklist: https://docs.nvidia.com/deeplearning/performance/dl-performance-convolutional/index.html#checklist
- Digital GTC talks of interest:
  - S21351: Scaling the Transformer Model Implementation in PyTorch Across Multiple Nodes
  - S21547: Rebalancing the Load: Profile-Guided Optimization of the NAMD Molecular Dynamics Program for Modern GPUs using Nsight Systems
  - S21565: Roofline Performance Model for HPC and Deep-Learning Applications
  - S41723: How to understand and Optimize Shared Memory Access using Nsight Compute
  - D4121: Latest Updates to CUDA Developer Tools

Exascale Computing
Microprocessor and Quantum Computing
AI and Language Computing
IoE, Dependable and Secure Computing
GenNext Applied Computing
10010111001100
Advanced Computing For Human Advancement
Thank You !