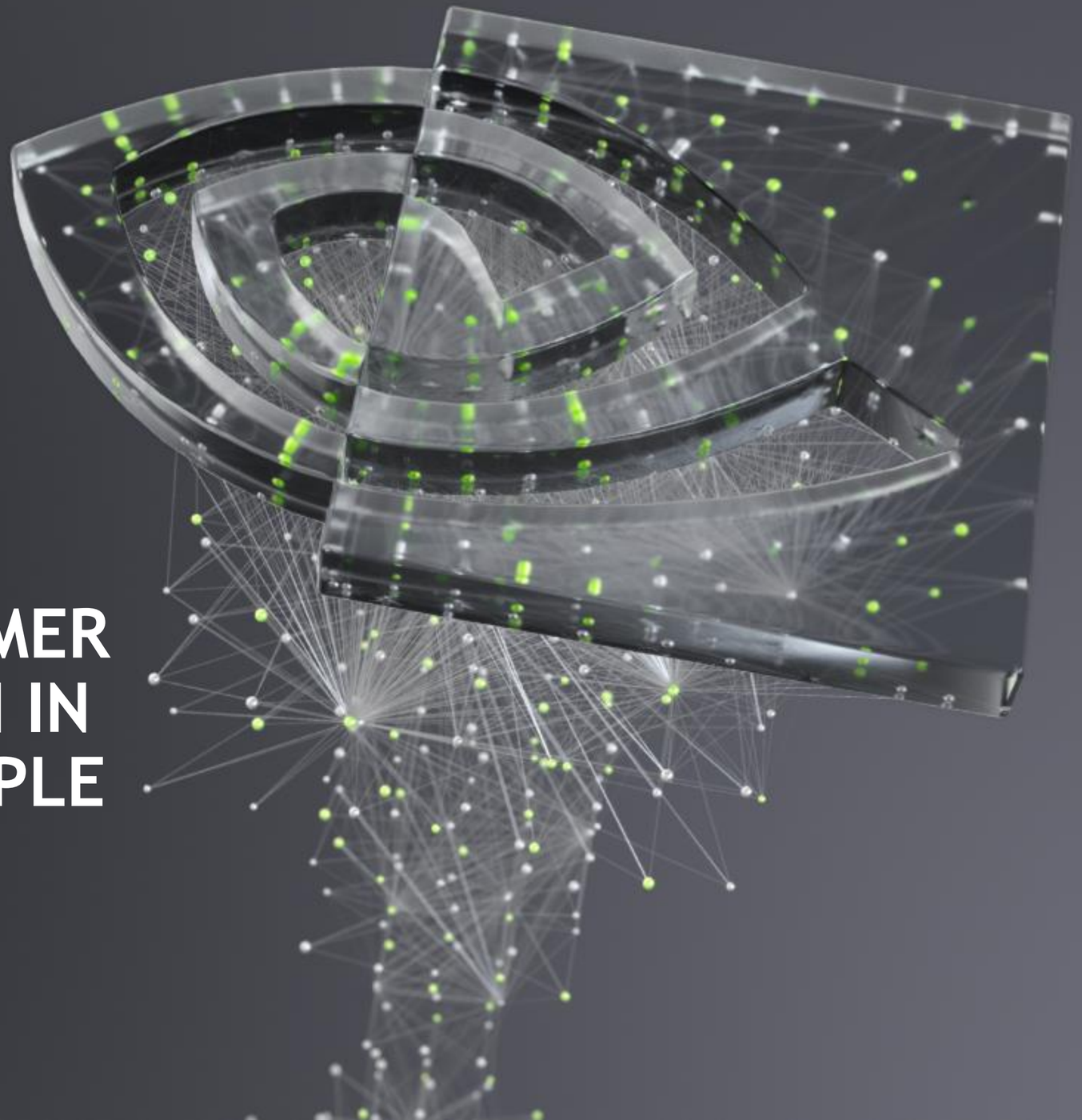


# SCALING THE TRANSFORMER MODEL IMPLEMENTATION IN PYTORCH ACROSS MULTIPLE NODES

Arslan Zulfiqar – Senior Architect, NVIDIA

Robert Knight – Software Engineer, NVIDIA

March 24, 2020

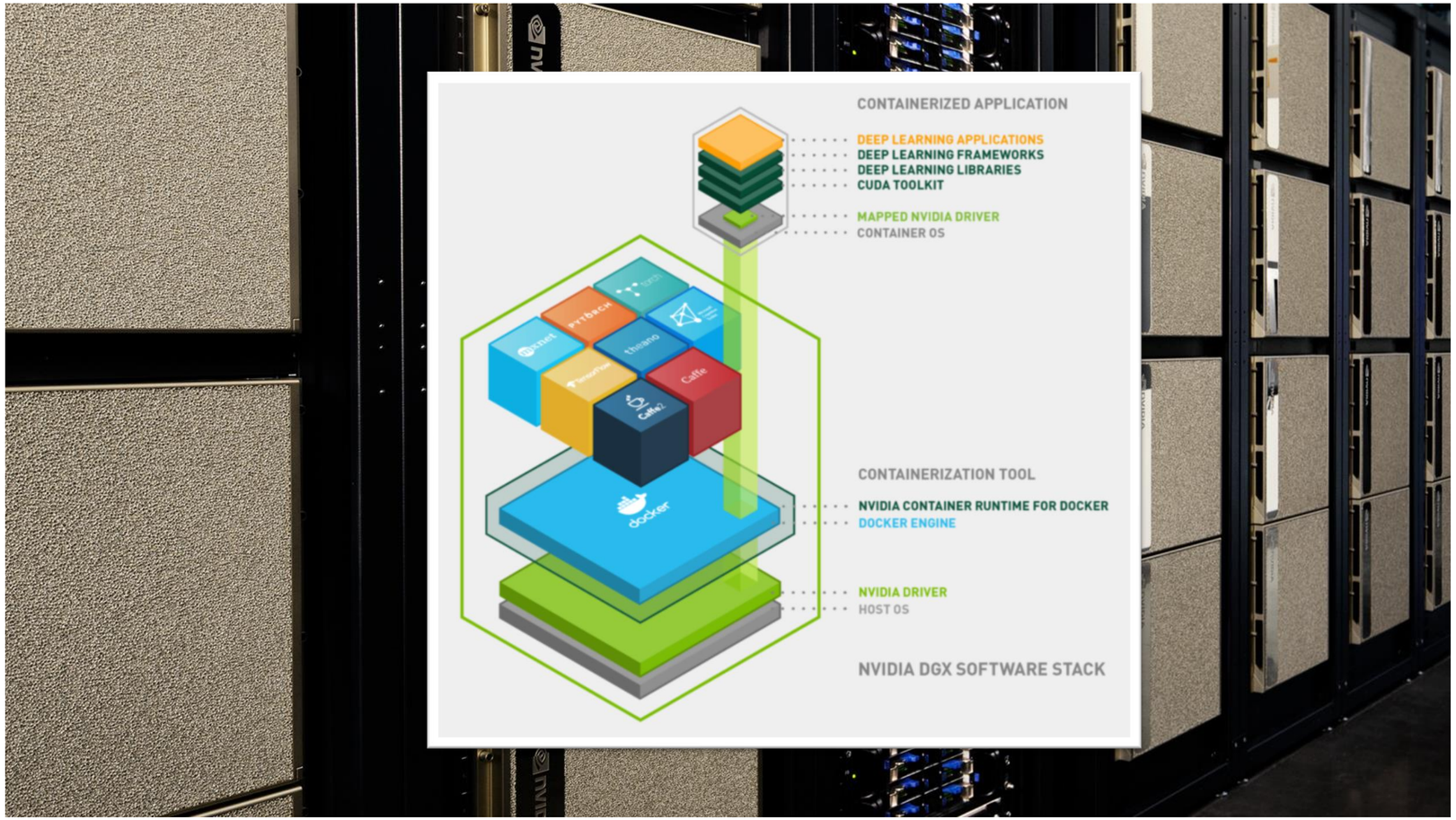
# SCALING WORKLOADS IN A DATA CENTER IS HARD

Performance Debug requires significant breadth

- Cluster hardware
- Cluster infrastructure
- Cluster software stack

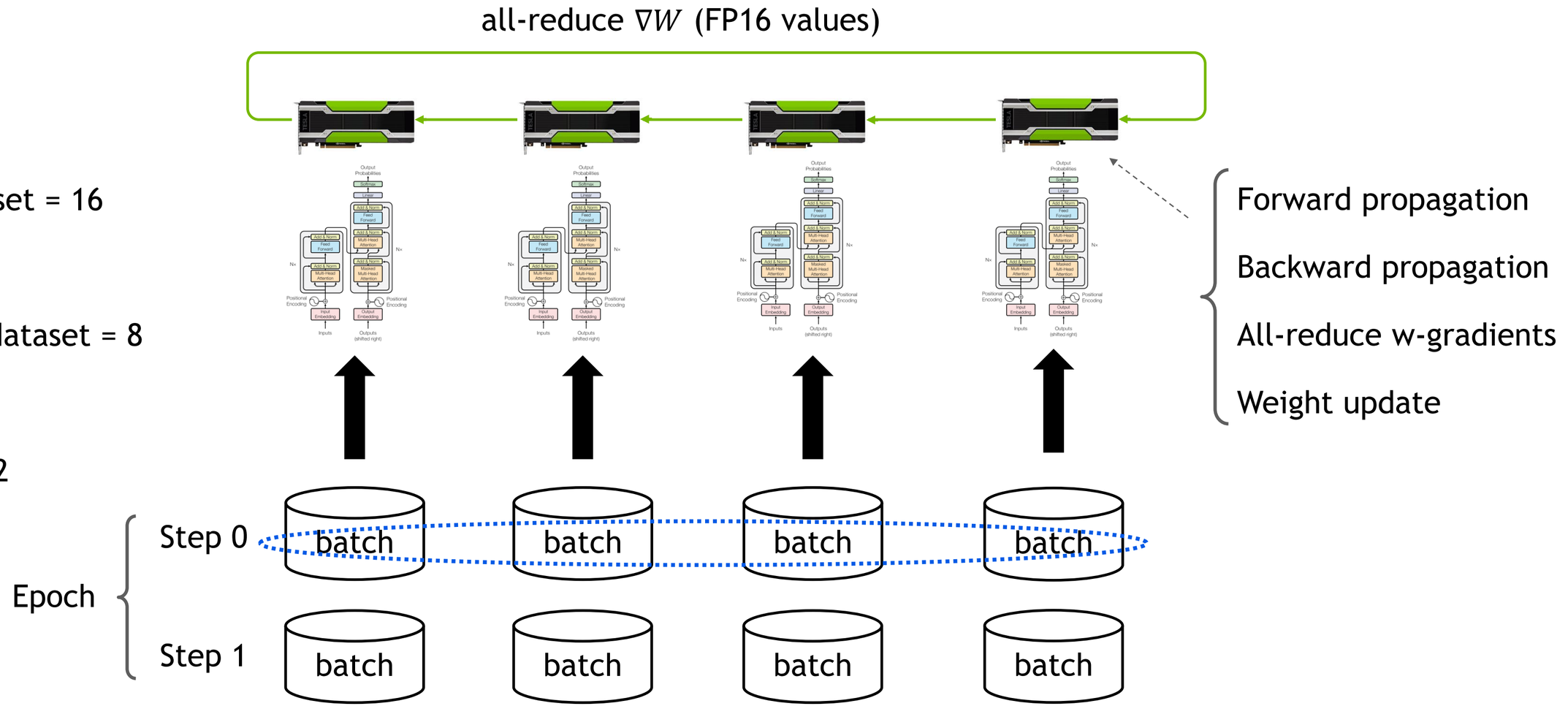
Visualizing Activity on 100s of Nodes Leads to Data Overload

Data Analytics Simplifies Debug



# DISTRIBUTED DEEP LEARNING TRAINING 101

## Most common scale-out technique: Data-parallel



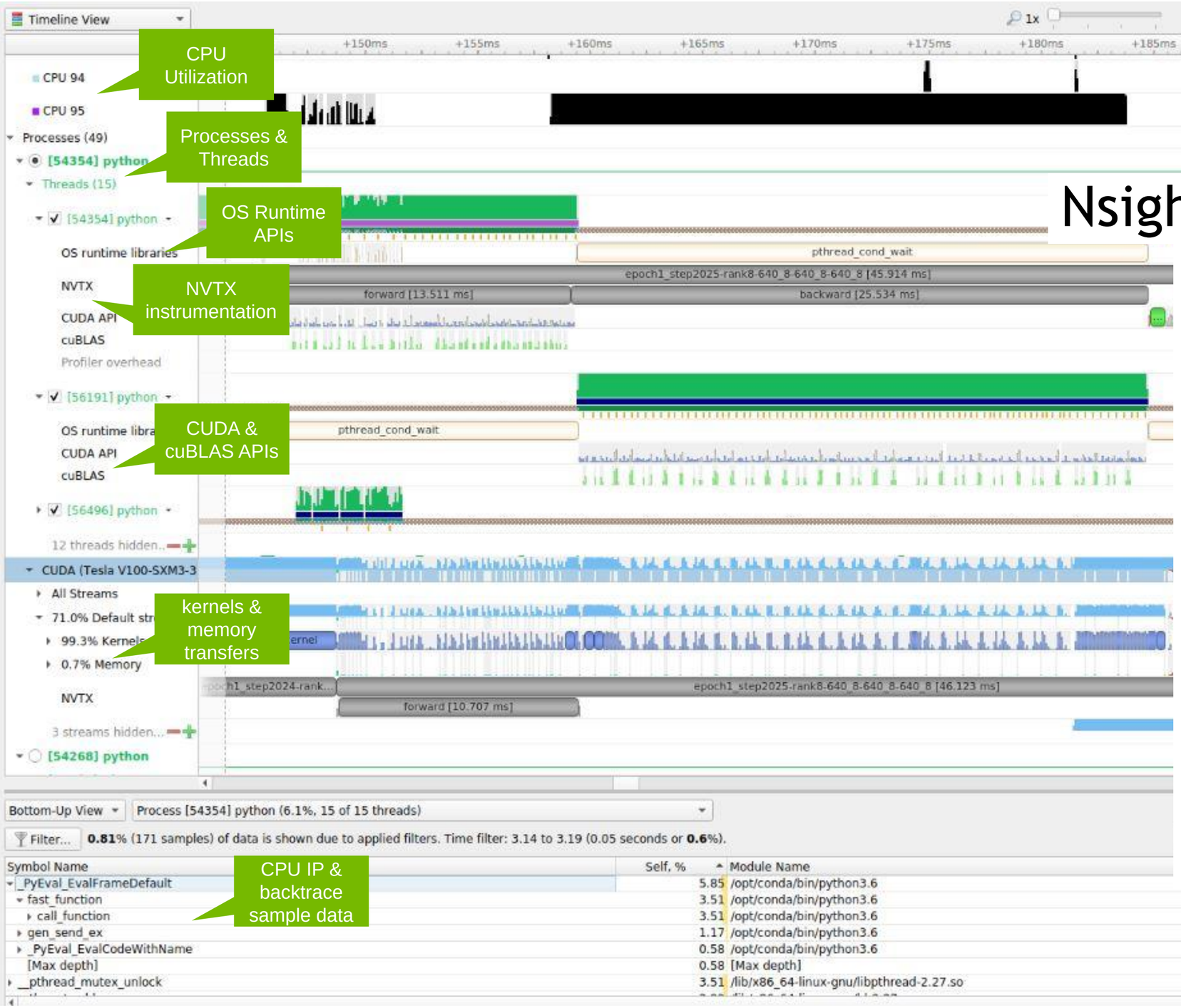
# DEEP LEARNING – RELAY RACING IN A DATA CENTER

Epoch = Race; Step = Race Segment

Each ‘team’ has 1000s of runners / segment (i.e. 1000s of processes / step)

A single Epoch is like a relay race with 1000s of runners (processes) per team per segment (step) of the race.
***Efficiency defined by the slowest runner***.

# NSIGHT SYSTEMS




# Nsight Systems Functionality

Traceable APIs
- CUDA
- cuDNN
- cuBLAS
- TensorRT
- OS Runtime
- MPI
- OpenMP
- OpenACC
- Vulkan
- OpenGL

GPU Context Switch

ftrace

Export Data To:
- SQLite
- HDF
- JSON
- Text

Architectures Supported:
- Power
- ARM Server
- Tegra
- X86_64

Linux

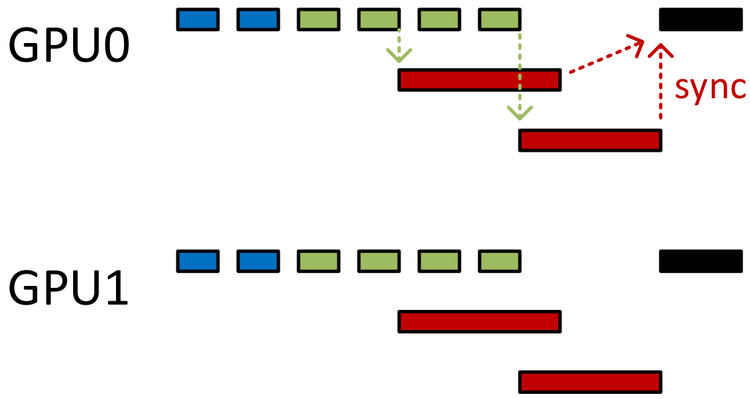


NVIDIA

# SCALE-OUT DEBUG

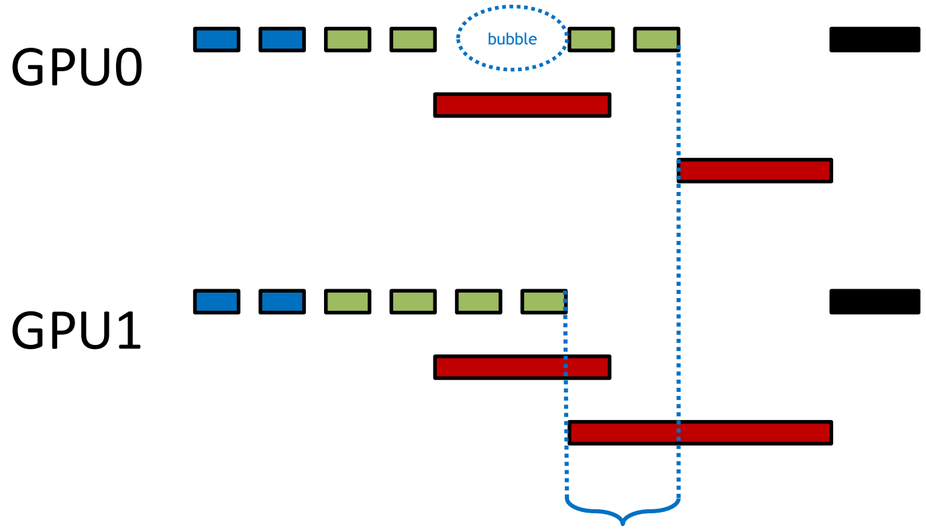
# VARIATIONS IN EXECUTION CAN LEAD TO EFFICIENCY LOSS

- Forward kernel
- Backward kernel
- NCCL all-reduce kernel
- Weight-upd. Kernel (optimizer)




Note: Timeline may look different if e.g. using:

- Model-parallelism
- Distributed Batch Normalization
- Distributed optimizer
- Async pre-processing side-streams




Jitter time overhead → efficiency loss!

This talk: Understand/ optimize efficiency loss due to variations in scale-out DL training via data analytics

Magnitude of efficiency loss due to jitter?

Which Step?
Which Node?
Which GPU?
When did efficiency start to diverge within a step?

Where do we look in profiles to isolate cause(s)?

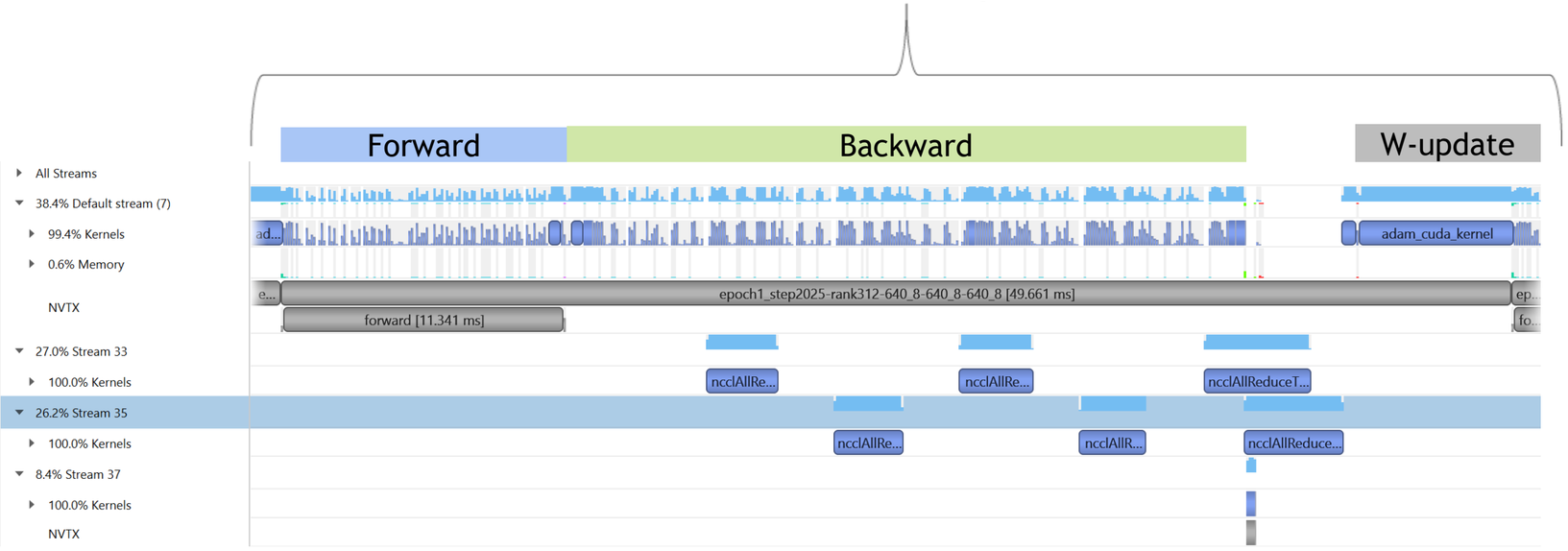
Culprit(s): CPU overheads, GPU overheads, both?

# CASE STUDY

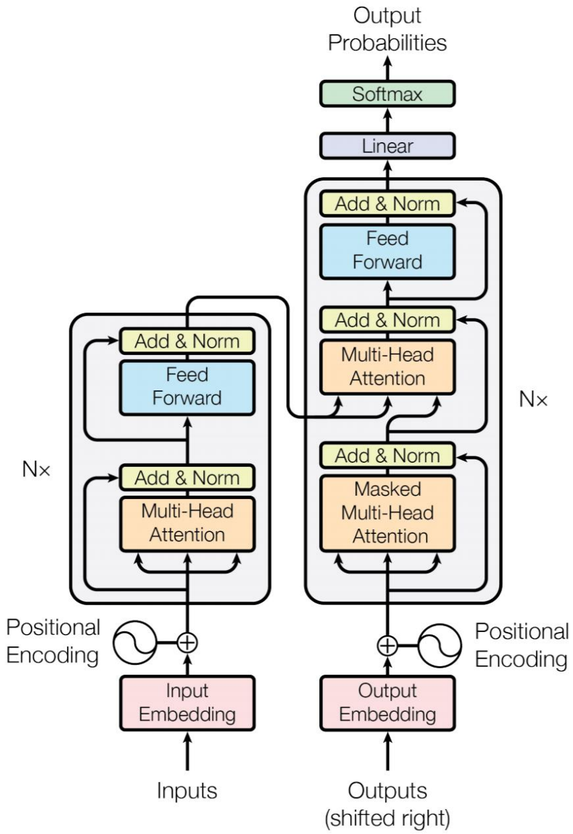
## Transformer Model in PyTorch

- 960 GPUs
- 903 kernels / step
- Batch/ GPU = 640 tokens (synthetic/ identical)
- 2000 warm-up steps; 50 profiling steps

### One Step

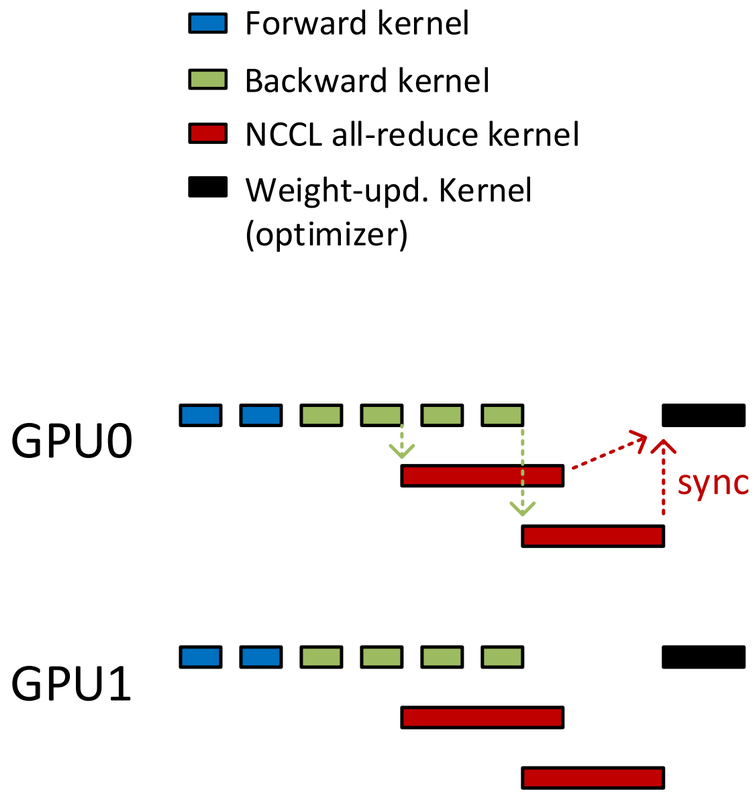



### Transformer Model



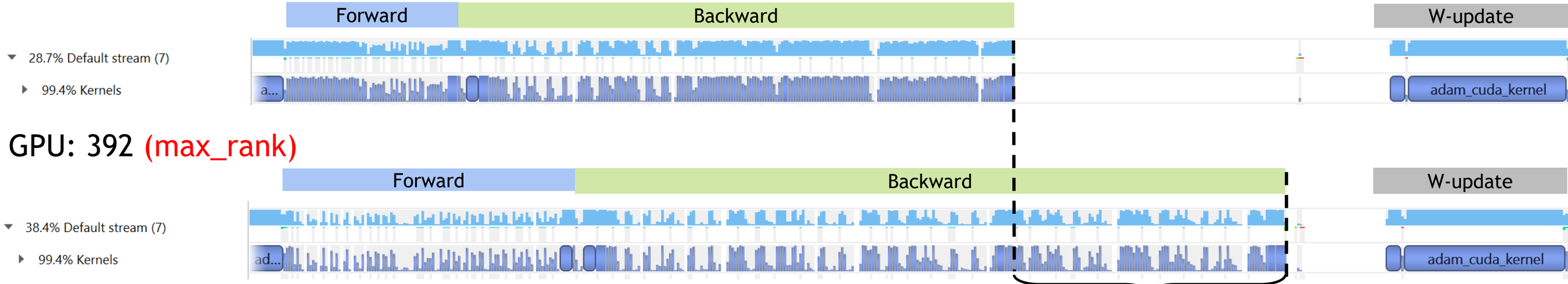
# OBSERVATIONS

- Every Step executes the same kernels using the same data
  - synthetic data used
- All steps synchronize across GPUs at the end of the last NCCL AllReduce kernel
- NCCL kernel order doesn’t matter since run concurrently on separate streams



# JITTER

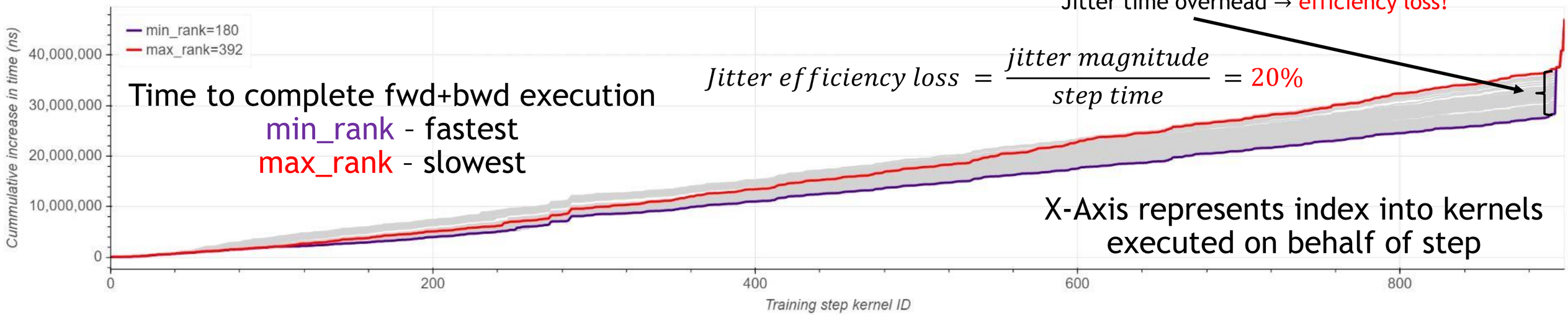
GPU: 180 (min_rank)




GPU: 392 (max_rank)




Jitter time overhead → efficiency loss!




Time to complete fwd+bwd execution
min_rank - fastest
max_rank - slowest

$$Jitter\ efficiency\ loss = \frac{jitter\ magnitude}{step\ time} = 20\%$$

X-Axis represents index into kernels executed on behalf of step



NVIDIA

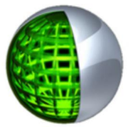
# BUILDING THE ANALYSIS

# TOOL TO ANALYZE SCALE-OUT DL TRAINING JITTER

## Uses popular, open-source Python packages

Task 1: Extract relevant profiler data

SQL



Nsight Systems

DBeaver

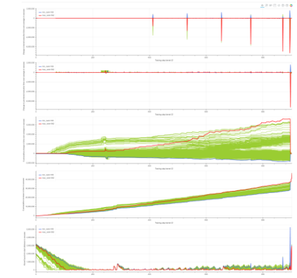


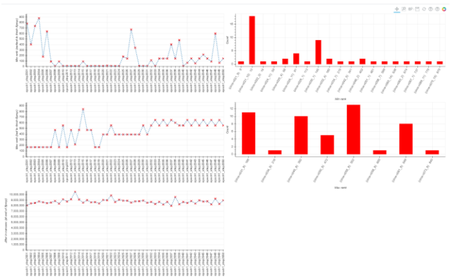
Dataframes

pandas

Visualizations

bokeh

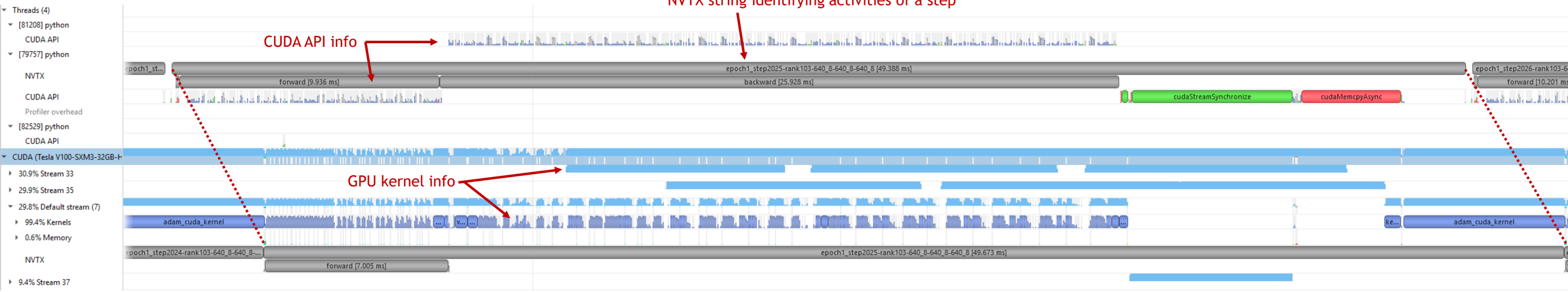




# EXTRACT RELEVANT PROFILER DATA

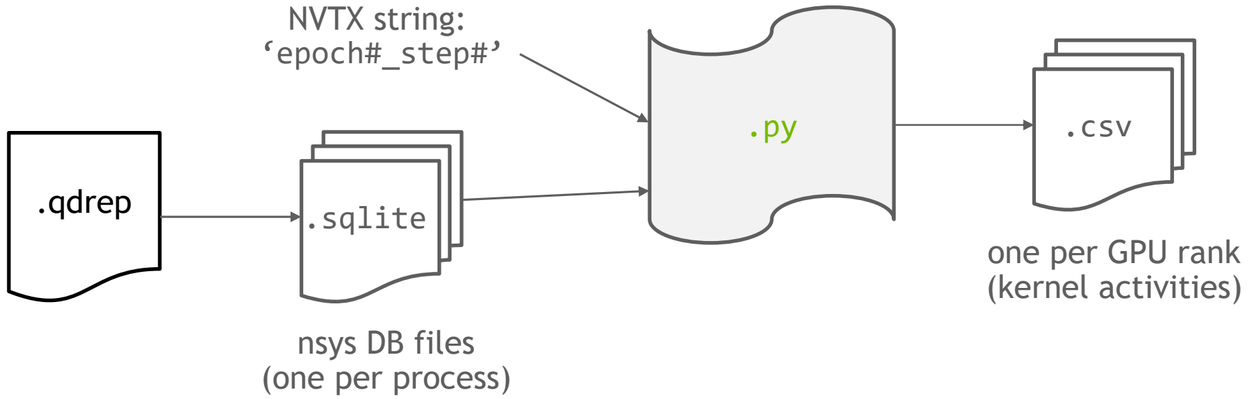
## Why? For analysis in our favorite tool (Pandas in our case)

Timeline of 1 GPU process of the multi-GPU run



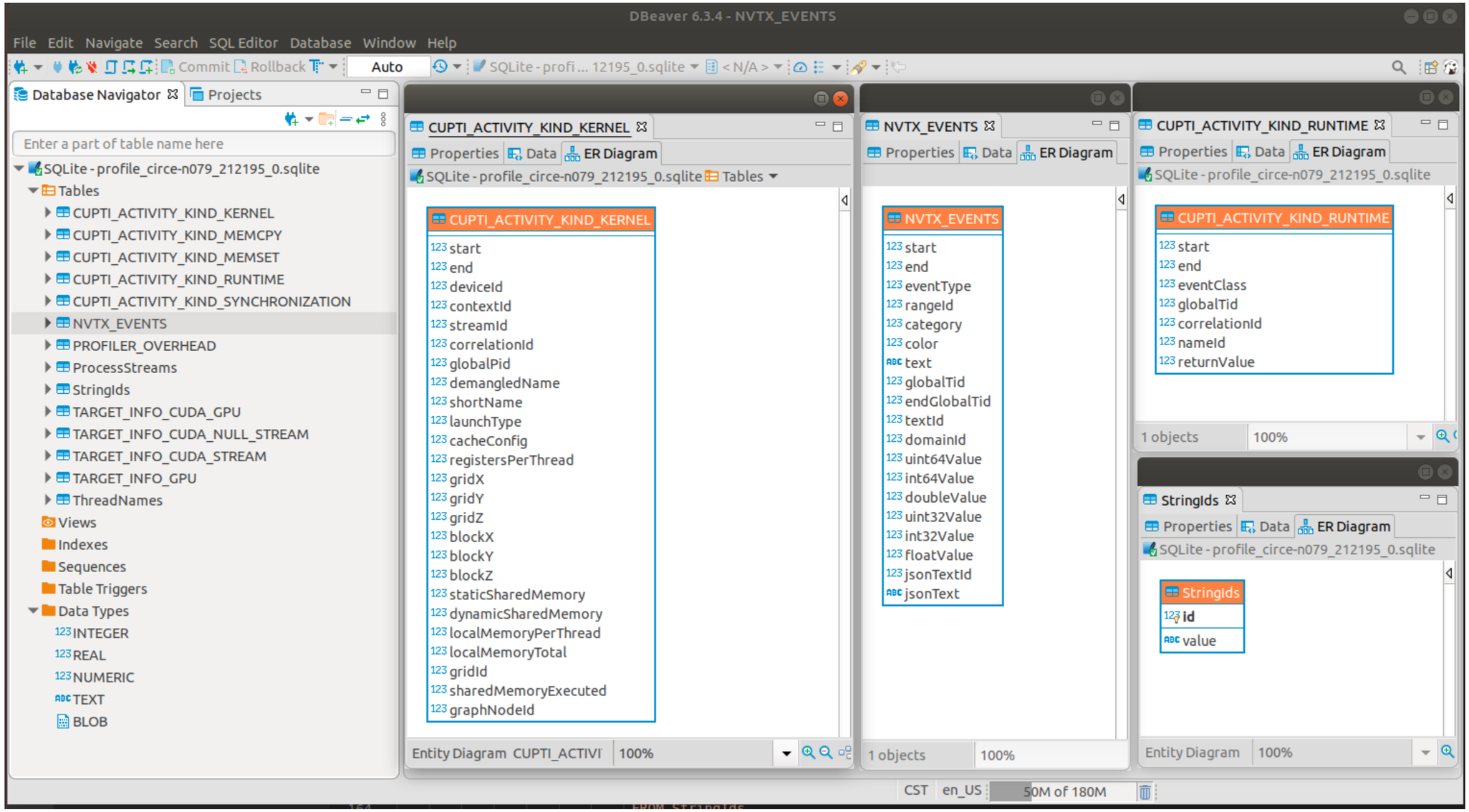

```
nsys profile -c cudaProfilerApi -t cuda,nvtx -s none

nsys export -t sqlite report1.qdrep
```



*Schema subject to change*
*in future versions of Nsight Systems*

# NSIGHT SYSTEMS EXPORT TO SQLITE



# EXTRACT RELEVANT PROFILER DATA 2

## One CSV per GPU per Step

| A | B | C | D | E | F | G | H | I | J | K | L |
|---|---|---|---|---|---|---|---|---|---|---|---|
| nvtx_text | nvtx_start | nvtx_end | device | crt_name | crt_start | crt_end | kernel_demangled_name | kernel stream | kernel_start | kernel_end | kernel_duration |
| epoch1_step2003-rank0-640_8-640_8-640_8 | 2078218693 | 2119940042 | 0 | cudaLaunchKernel_v7000 | 2078744304 | 2078762194 | void indexSelectLargeIndex<c10::Ha | 7 | 2081895540 | 2081903732 | 8192 |
| epoch1_step2003-rank0-640_8-640_8-640_8 | 2078218693 | 2119940042 | 0 | cudaLaunchKernel_v7000 | 2078799192 | 2078809641 | void at::native::elementwise_kernel< | 7 | 2081904724 | 2081908692 | 3968 |
| epoch1_step2003-rank0-640_8-640_8-640_8 | 2078218693 | 2119940042 | 0 | cudaLaunchKernel_v7000 | 2078858614 | 2078868767 | void at::native::elementwise_kernel< | 7 | 2081910036 | 2081911924 | 1888 |
| epoch1_step2003-rank0-640_8-640_8-640_8 | 2078218693 | 2119940042 | 0 | cudaLaunchKernel_v7000 | 2078883166 | 2078892114 | void at::native::elementwise_kernel< | 7 | 2081912948 | 2081914836 | 1888 |
| epoch1_step2003-rank0-640_8-640_8-640_8 | 2078218693 | 2119940042 | 0 | cudaLaunchKernel_v7000 | 2078910397 | 2078918709 | void THCTensor_kernel_scanInnerm | 7 | 2081916660 | 2081921140 | 4480 |
| epoch1_step2003-rank0-640_8-640_8-640_8 | 2078218693 | 2119940042 | 0 | cudaLaunchKernel_v7000 | 2078931166 | 2078938950 | void at::native::elementwise_kernel< | 7 | 2081922068 | 2081923444 | 1376 |
| epoch1_step2003-rank0-640_8-640_8-640_8 | 2078218693 | 2119940042 | 0 | cudaLaunchKernel_v7000 | 2078960580 | 2078967487 | void at::native::elementwise_kernel< | 7 | 2081924436 | 2081925684 | 1248 |
| epoch1_step2003-rank0-640_8-640_8-640_8 | 2078218693 | 2119940042 | 0 | cudaLaunchKernel_v7000 | 2078991832 | 2078999976 | void indexSelectLargeIndex<c10::Ha | 7 | 2081926420 | 2081933620 | 7200 |
| epoch1_step2003-rank0-640_8-640_8-640_8 | 2078218693 | 2119940042 | 0 | cudaLaunchKernel_v7000 | 2079022018 | 2079029246 | void at::native::elementwise_kernel< | 7 | 2081934580 | 2081937652 | 3072 |
| epoch1_step2003-rank0-640_8-640_8-640_8 | 2078218693 | 2119940042 | 0 | cudaLaunchKernel_v7000 | 2079061049 | 2079069674 | void at::native::(anonymous namesp | 7 | 2081938580 | 2081944597 | 6017 |

Lists all kernel executed within a single step

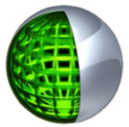
Correlations between steps, kernel launches, kernel executions established

# TOOL TO ANALYZE SCALE-OUT DL TRAINING JITTER

## Uses popular, open-source Python packages

Task 2: Calculate analysis data

SQL



Nsight Systems

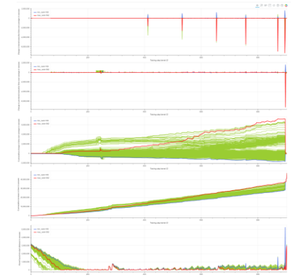
DBeaver

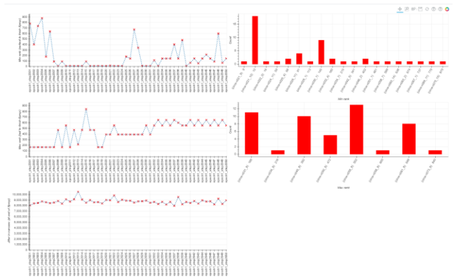
Dataframes

pandas

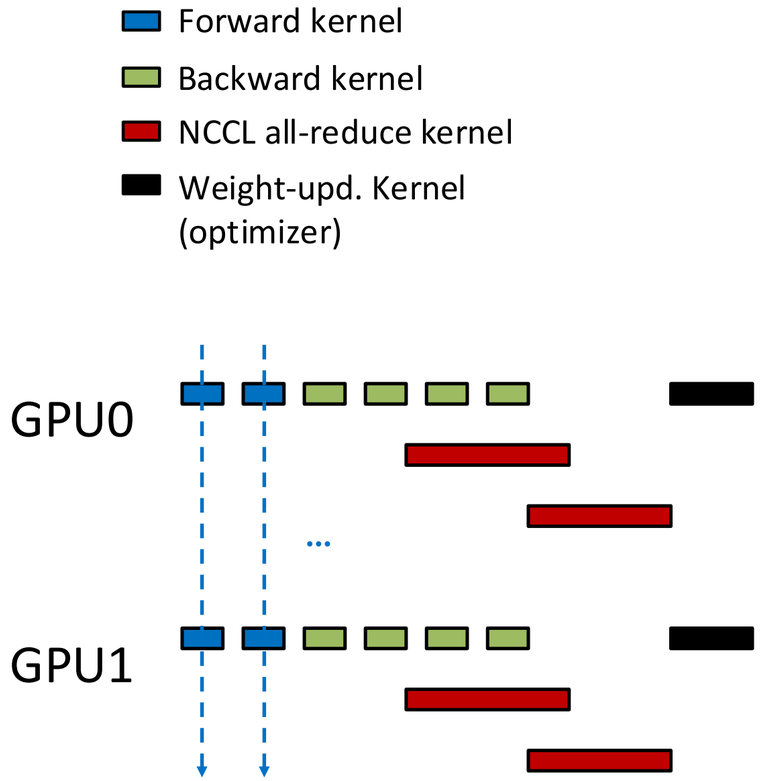
Visualizations

bokeh





# KERNEL DURATIONS




Calculate mean across GPUs for each kernel

Calculate 2 Sets of values (using Pandas dataframes)

- Kernel Duration (nanoseconds)

| Kernel Index | Kernel Name | GPU 0 | GPU 1 | GPU … | GPU 959 |
|---|---|---|---|---|---|
| 0 | a | 9176 | 10575 | 7866 | 8234 |
| 1 | b | 3451 | 6456 | 9765 | 4401 |
| … | … | … | … | … | … |
| 902 | b | 4144 | 7345 | 5721 | 5177 |

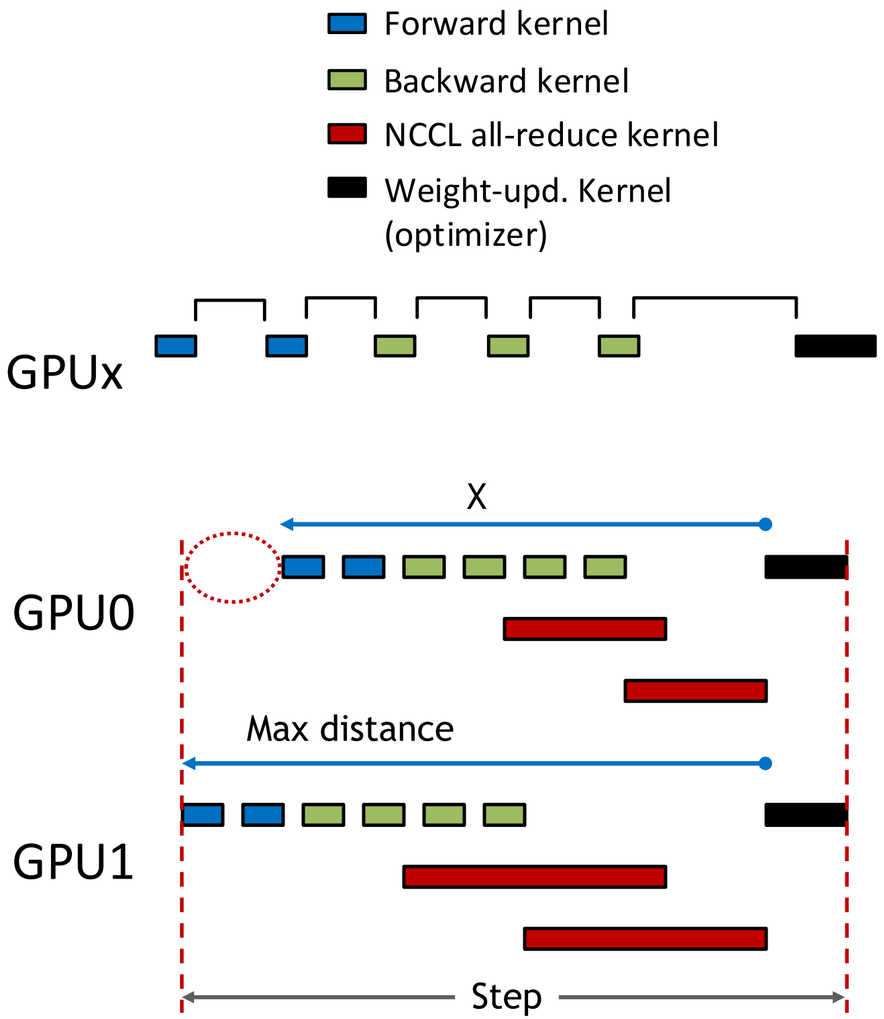
- Kernel Duration – Average Kernel Duration (computed across GPUs)

| Kernel Index | Kernel Name | GPU 0 | GPU 1 | GPU … | GPU 959 |
|---|---|---|---|---|---|
| 0 | a | -435 | 964 | -1745 | -1377 |
| 1 | b | -2111 | 894 | 4203 | 1161 |
| … | … | … | … | … | … |
| 902 | b | -1086 | 2115 | 491 | -53 |

| Average Kernel Duration |
|---|
| 9611 |
| 5562 |
| … |
| 5230 |

-435=9176-9611

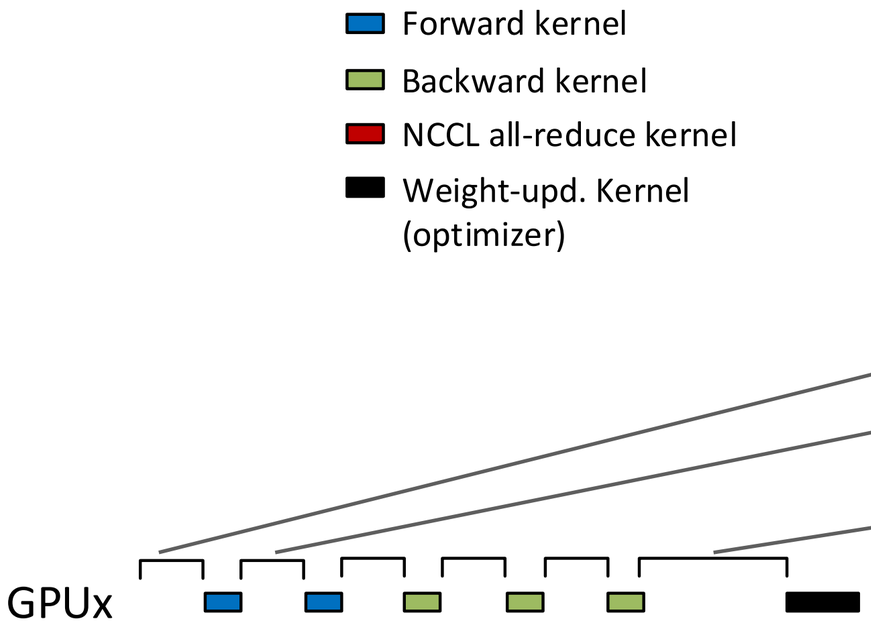
# KERNEL GAPS (TIME BETWEEN KERNELS)




Need to handle 3 cases:

1. Default stream kernels ***except the first one***
   - Subtract the previous kernel's end time from the current kernel's start time
2. Handle the first kernel's gap (red bubble)
   - Calculate distance X from the end of the last NCCL kernel to the start of kernel 0 (blue arrow)
   - Subtract calculated distance X from 'Max distance'
3. NCCL kernels use kernel gap of 0 (default stream kernel timestamps capture all the important interactions)

# KERNEL GAPS




Calculate 2 Sets of values (using Pandas dataframes)

- Kernel Gap (nanoseconds)

| Kernel Index | Kernel Name | GPU 0 | GPU 1 | GPU … | GPU 959 |
|---|---|---|---|---|---|
| 0 | a | 12921 | 11034 | 14548 | 9813 |
| 1 | b | 80124 | 94890 | 100344 | 91209 |
| … | … | … | … | … | … |
| 902 | b | 210467 | 197712 | 215983 | 200334 |

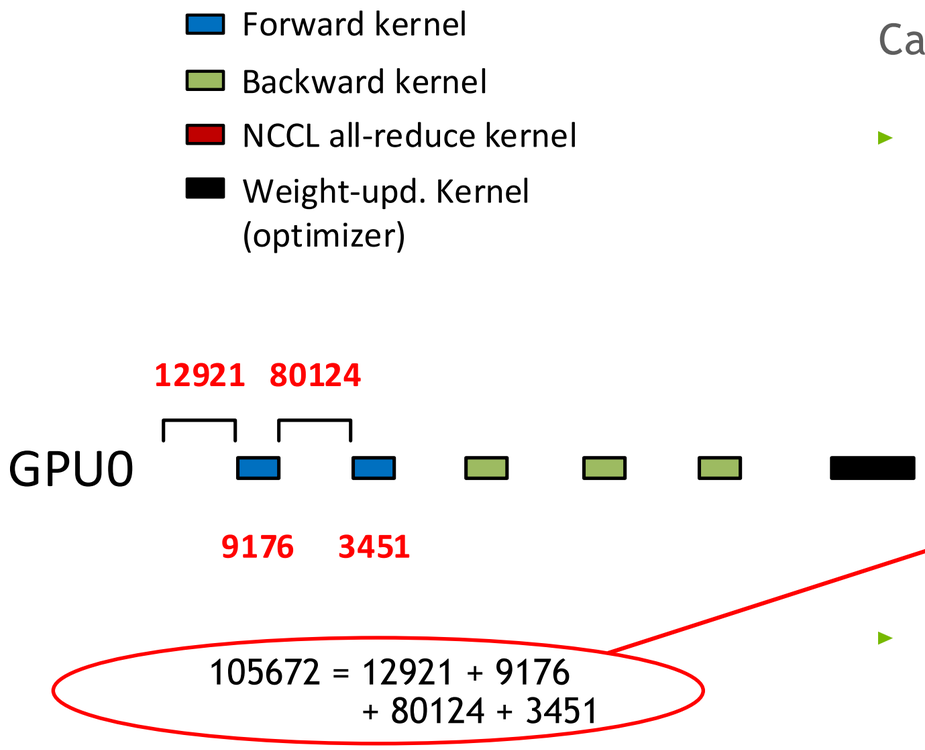
- Kernel Gap – Average Kernel Gap (computed across GPUs)

| Kernel Index | Kernel Name | GPU 0 | GPU 1 | GPU … | GPU 959 |
|---|---|---|---|---|---|
| 0 | a | 1377 | -510 | 3004 | -1731 |
| 1 | b | -17265 | -2499 | 2955 | -6180 |
| … | … | … | … | … | … |
| 902 | b | 8068 | -4687 | 13584 | -2065 |

| Average Kernel Gap |
|---|
| 11544 |
| 97389 |
| … |
| 202399 |

1377=12921-11544

# TIME TO KERNEL END




Calculate 2 Sets of values (using Pandas dataframes)

- Time to Kernel End

| Kernel Index | Kernel Name | GPU 0 | GPU 1 | GPU … | GPU 959 |
|---|---|---|---|---|---|
| 0 | a | 22097 | 21609 | 22414 | 18047 |
| 1 | b | 105672 | 122415 | 132523 | 113657 |
| … | … | … | … | … | … |
| 902 | b | … | … | … | … |

- Time to Kernel End – Time to Kernel End (computed across GPUs)

| Kernel Index | Kernel Name | GPU 0 | GPU 1 | GPU … | GPU 959 |
|---|---|---|---|---|---|
| 0 | a | -1469 | -1957 | -1152 | -5519 |
| 1 | b | -14139 | 2604 | 12712 | -6154 |
| … | … | -… | … | … | … |
| 902 | b | … | … | … | … |

| Average Time to Kernel End |
|---|
| 23566 |
| 119811 |
| … |
| … |

# SUMMARY OF CALCULATED DATA

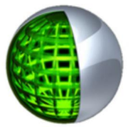
- Kernel durations across GPUs (*find long kernels, load imbalances*)
- Kernel gaps across GPUs (*find CPU issues – e.g. thread synchronization*)
- Cumulative Time to Kernel End across GPUs (ensure NCCL kernels 0'ed out before applying)

- Note: Raw kernel durations were obtained from profiler data. All other table data was calculated.

# TOOL TO ANALYZE SCALE-OUT DL TRAINING JITTER

## Uses popular, open-source Python packages

SQL



Nsight Systems

DBeaver

Dataframes

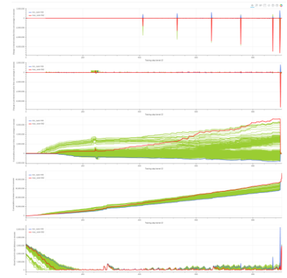
pandas

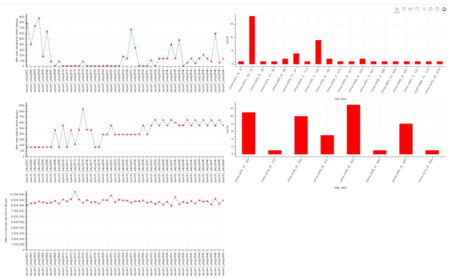
Task 3: Visualize

Visualizations

bokeh

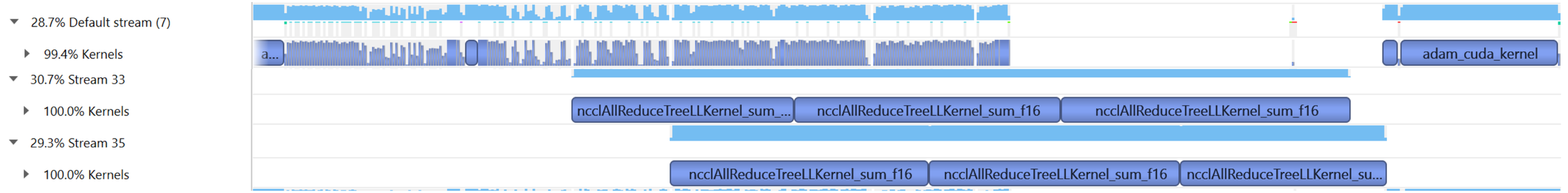
Use your favorite package



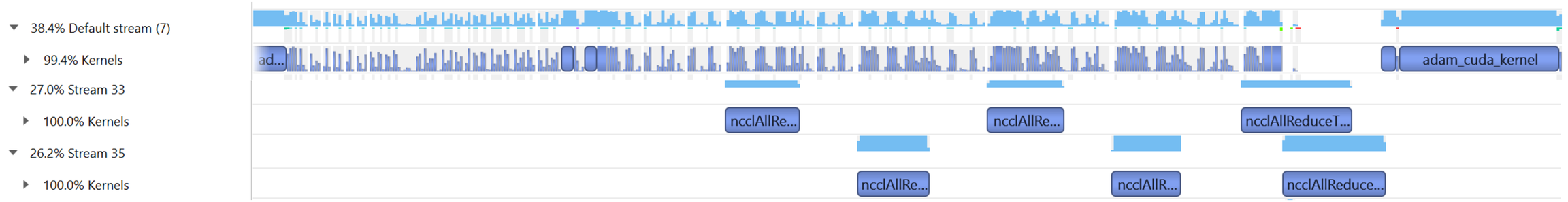


# KERNEL DURATION VS AVERAGE KERNEL DURATION

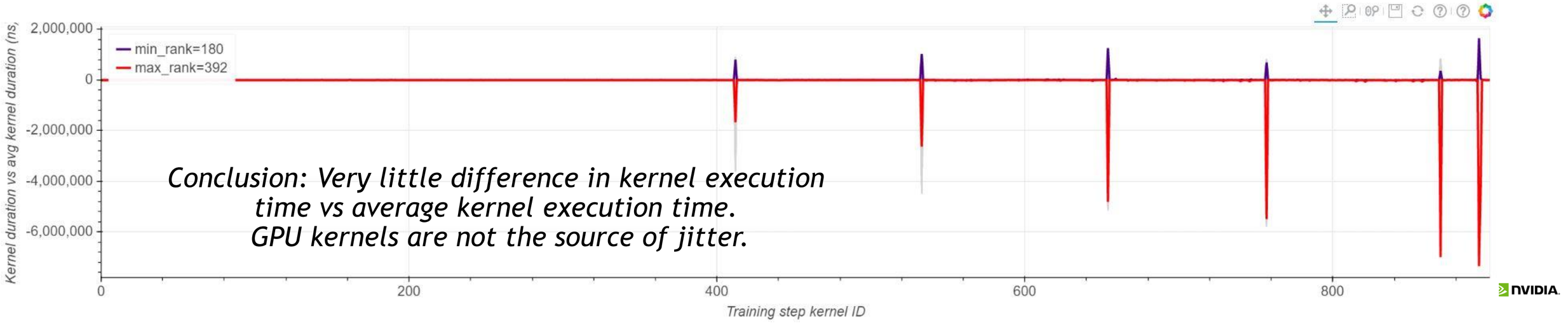
GPU: 180
(min_rank)




GPU: 392
(max_rank)



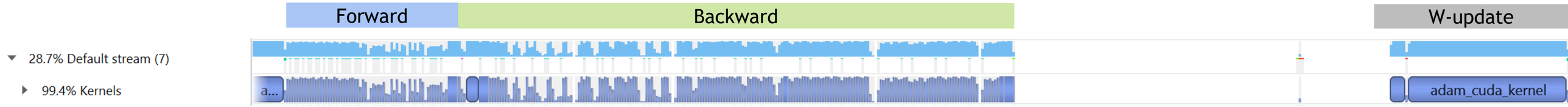

GPU 180: 6 NCCL kernels forced to wait causes purple spikes  GPU 392: NCCL kernels run fast (no waiting) causes red spikes



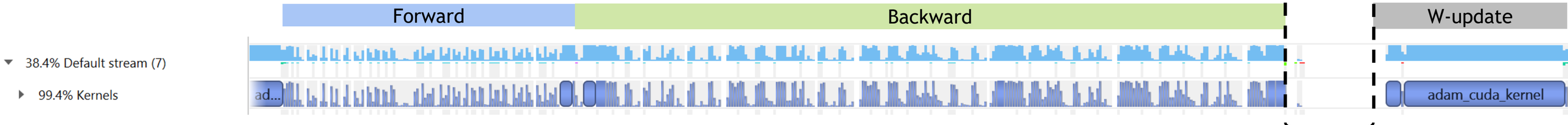

*Conclusion: Very little difference in kernel execution time vs average kernel execution time. GPU kernels are not the source of jitter.*

# KERNEL GAP VS AVERAGE KERNEL GAP

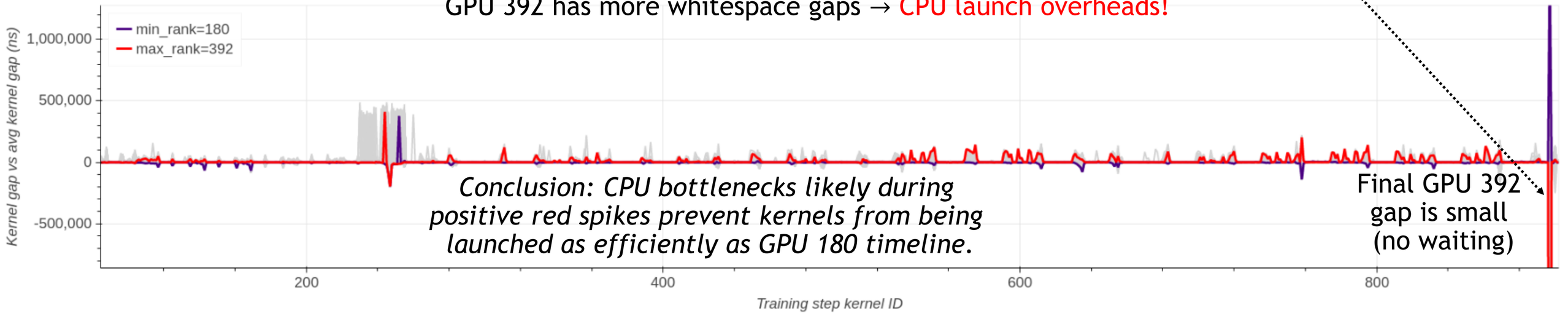
GPU: 180 (min_rank)




GPU: 392 (max_rank)



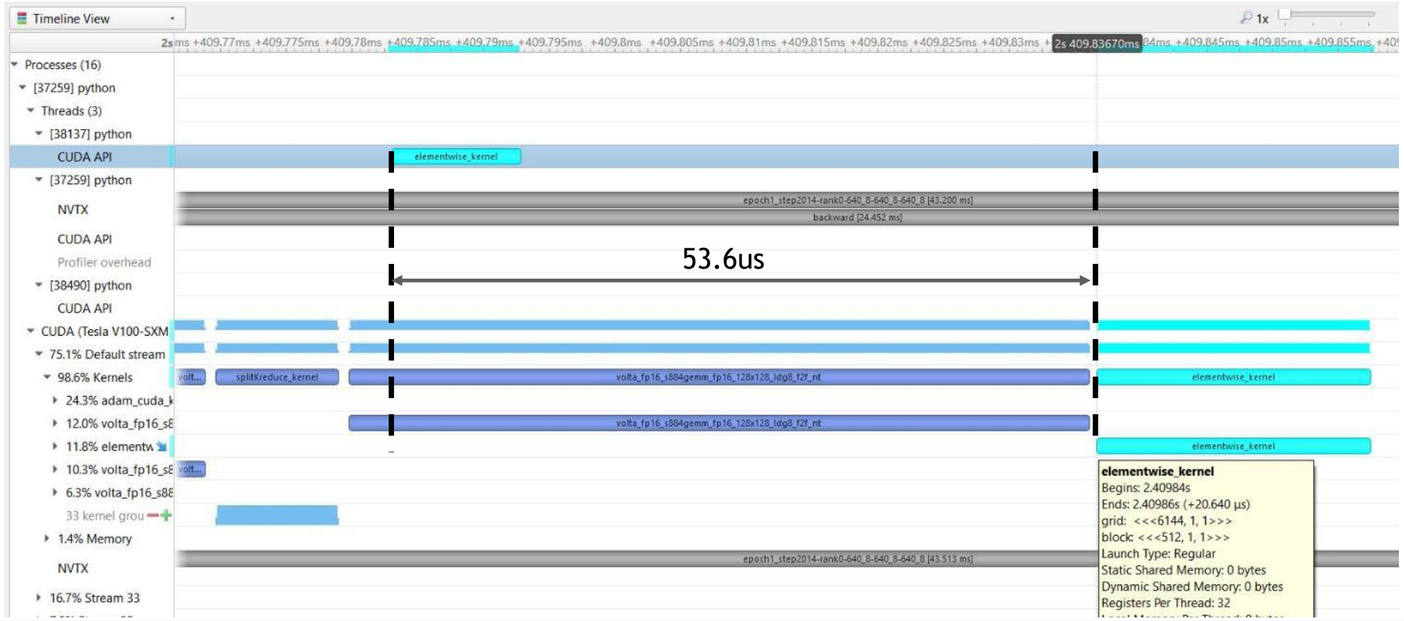

GPU 392 has more whitespace gaps → CPU launch overheads!



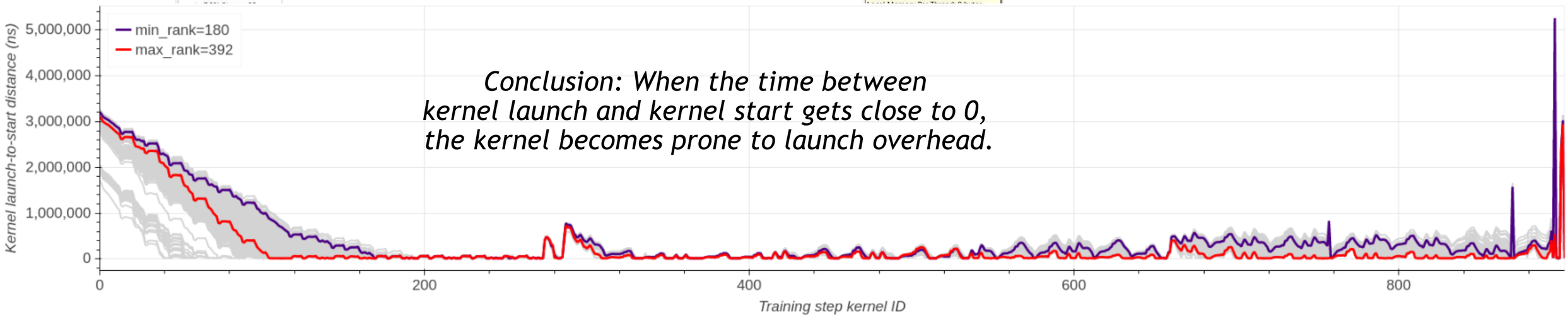

*Conclusion: CPU bottlenecks likely during positive red spikes prevent kernels from being launched as efficiently as GPU 180 timeline.*

Final GPU 392 gap is small (no waiting)

# KERNEL LAUNCH TO GPU EXECUTION START




- On GPU 392, after kernel ~100, CPU falls behind (kernels launched immediately after being scheduled)
  - Allows for gaps to build in GPU timeline.




*Conclusion: When the time between kernel launch and kernel start gets close to 0, the kernel becomes prone to launch overhead.*

# TIME TO KERNEL END VS. AVERAGE TIME TO KERNEL END

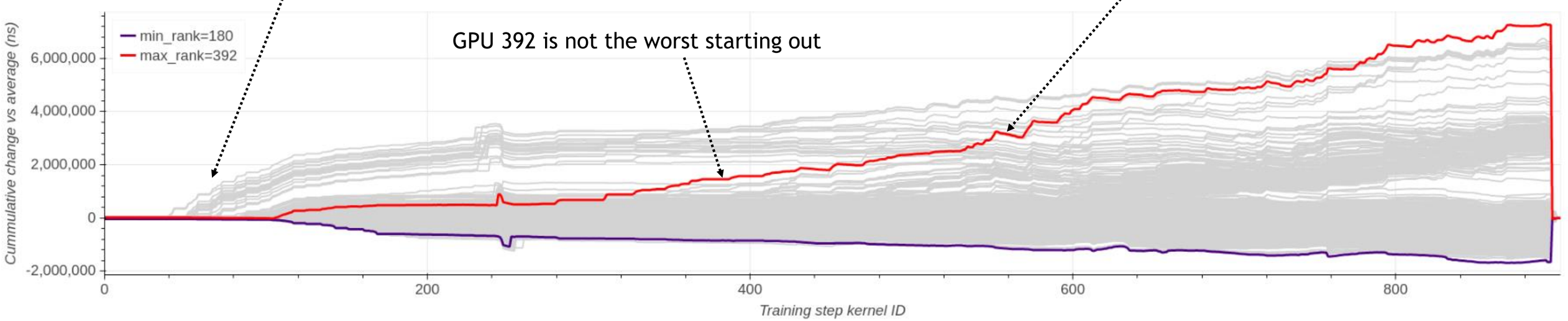
X-Axis represents index into kernels executed on behalf of step

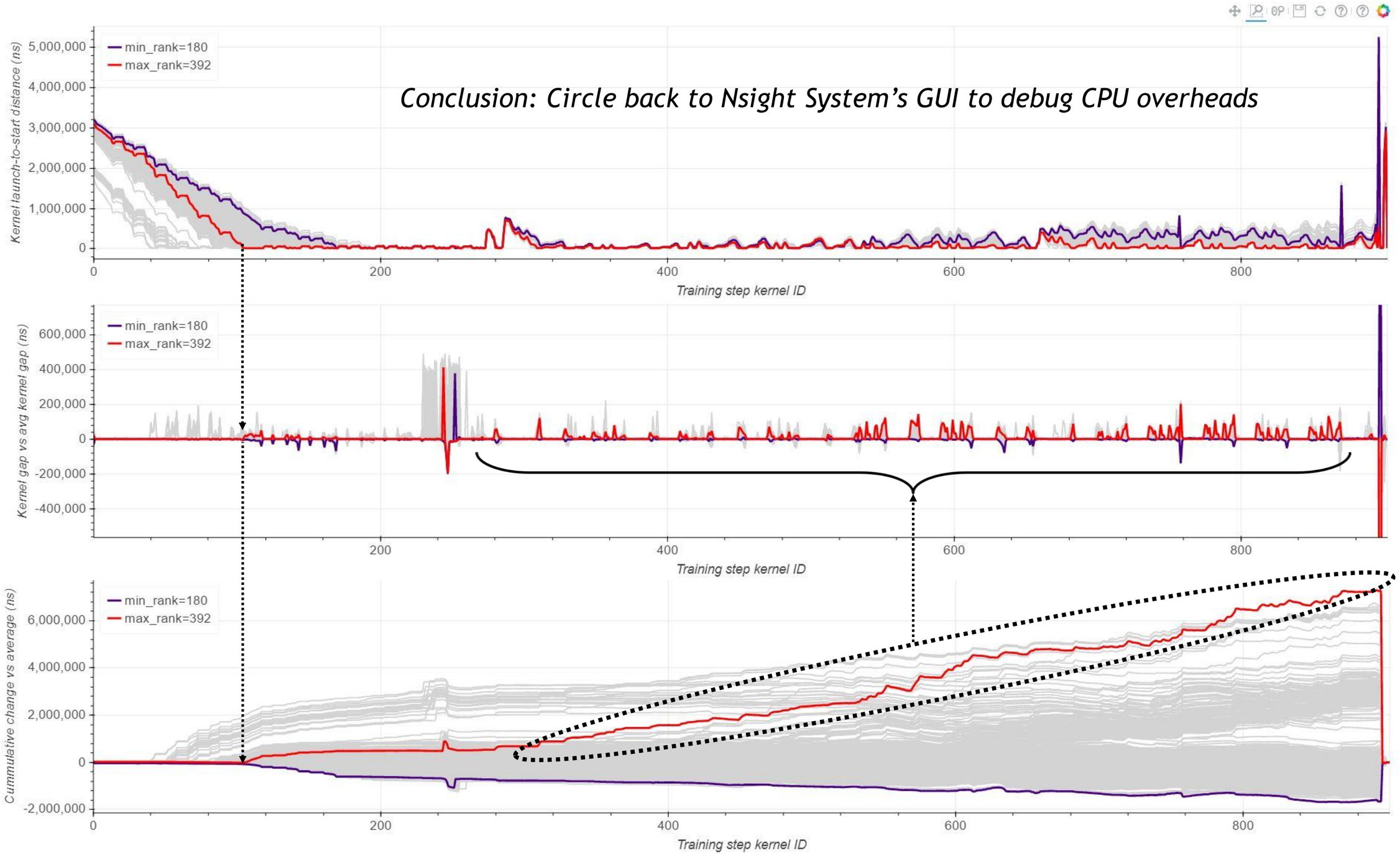
Y-Axis represents time divergence vs. average time to arrive at specific kernel end

Multiple GPUs start to diverge from average

For example: it took ~2.25ms longer than the average to arrive at the end of kernel ~560 on GPU 392.

GPU 392 is not the worst starting out

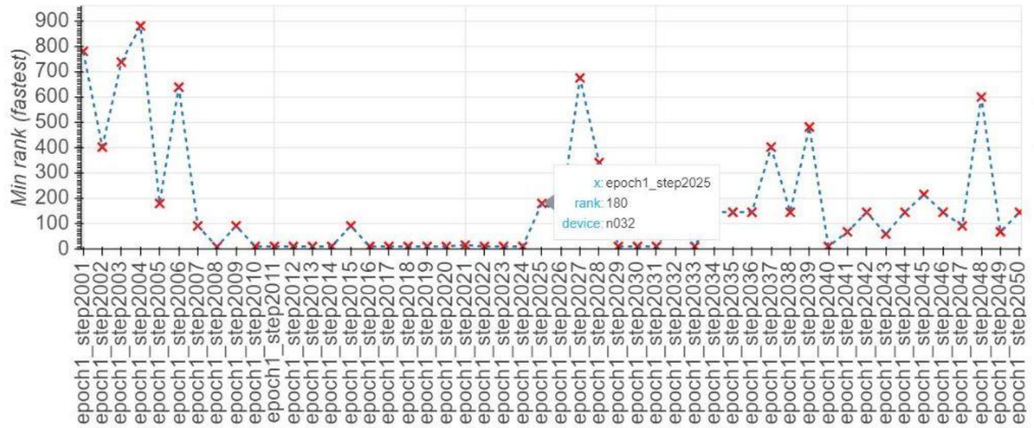



Conclusion: Circle back to Nsight System's GUI to debug CPU overheads
min_rank=180
max_rank=392
Kernel launch-to-start distance (ns)
5,000,000
4,000,000
3,000,000
2,000,000
1,000,000
0
0
200
400
600
800
Training step kernel ID
min_rank=180
max_rank=392
Kernel gap vs avg kernel gap (ns)
600,000
400,000
200,000
0
-200,000
-400,000
200
400
600
800
Training step kernel ID
min_rank=180
max_rank=392
Cummulative change vs average (ns)
6,000,000
4,000,000
2,000,000
0
-2,000,000
0
200
400
600
800
Training step kernel ID

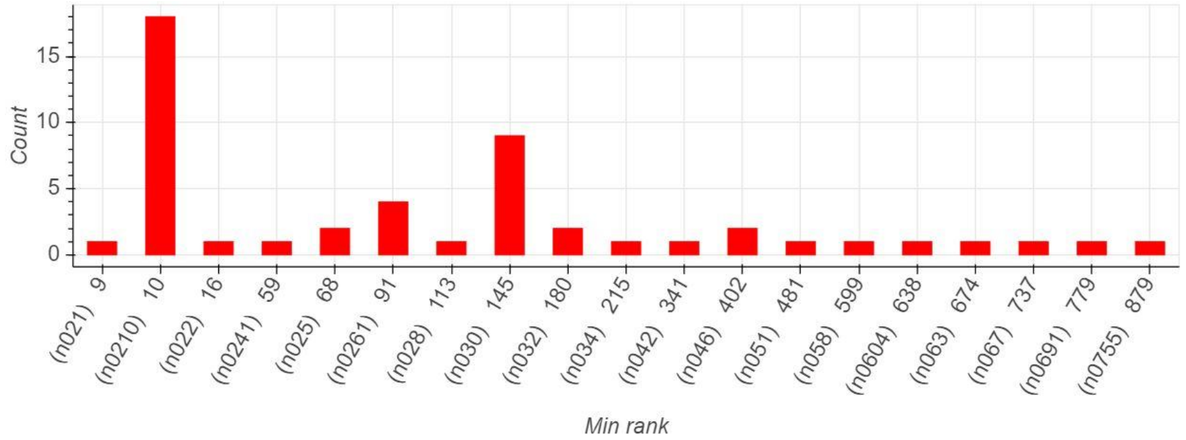
# EXTEND ANALYSIS TO SUMMARIZE IMPORTANT DATA

Extract relevant data from each step to show full profiling run behavior

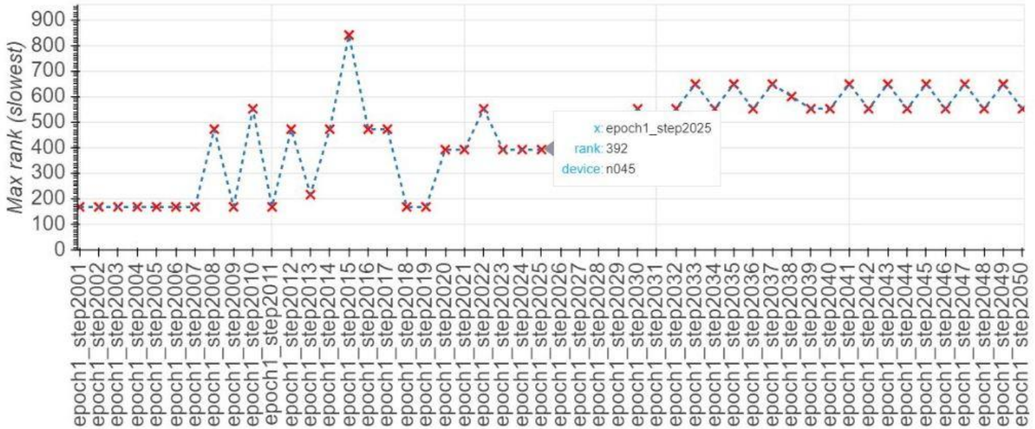
## Min Rank



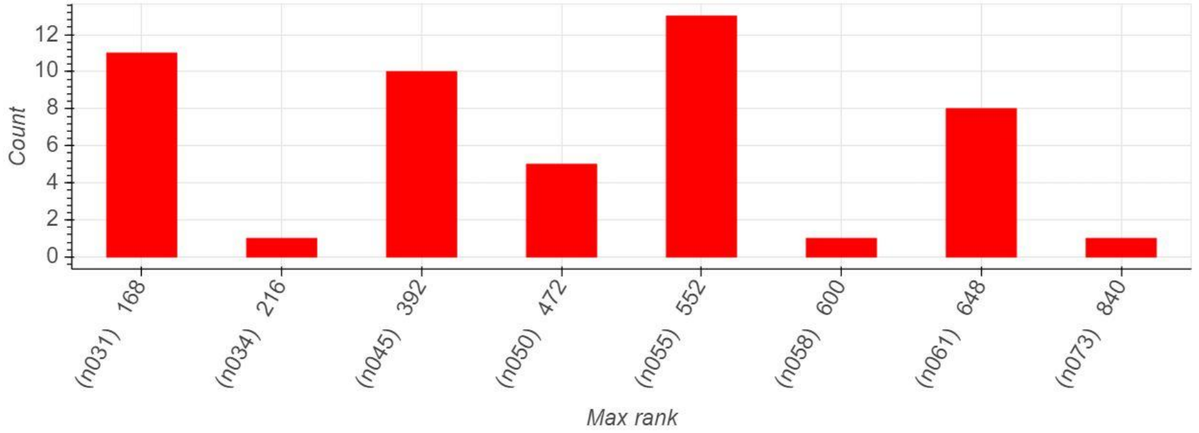

## Min Rank




## Max Rank




## Max Rank





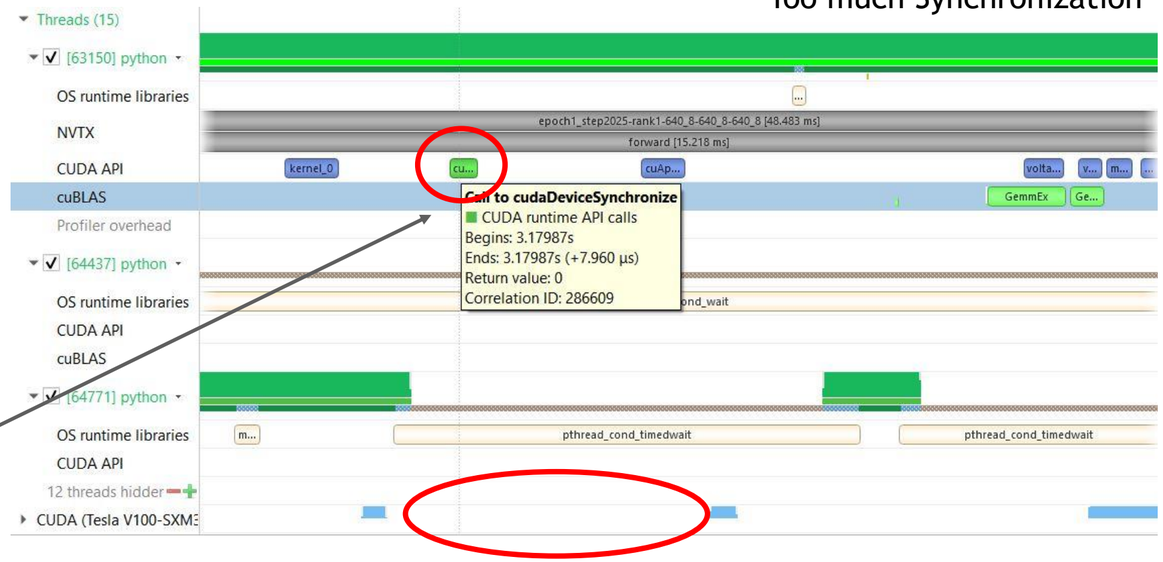

NVIDIA

# DEBUGGING ISSUES

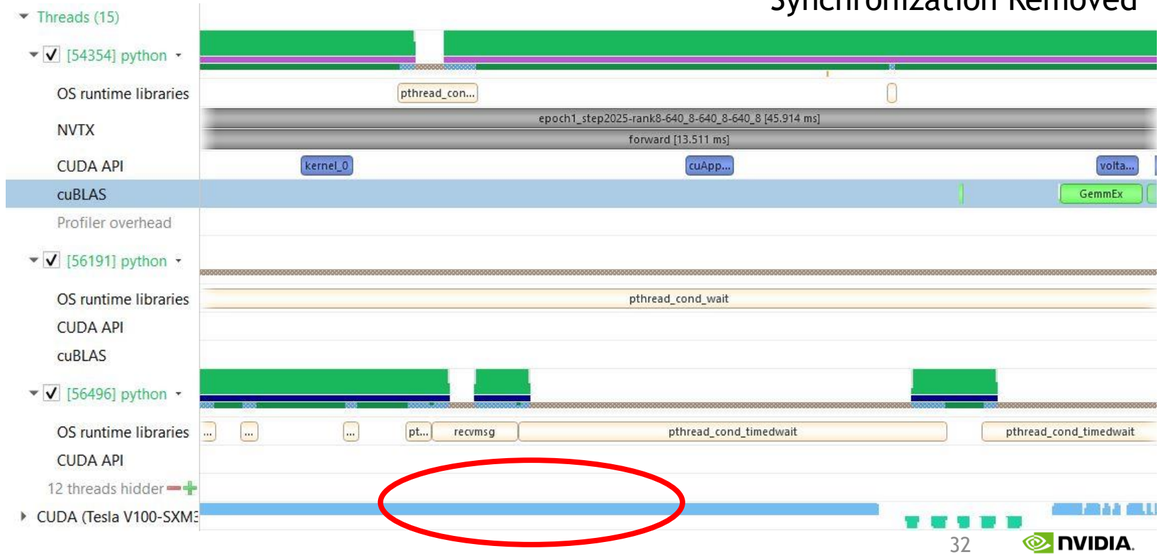
# PROFILER-GUIDED DEBUGGING

- CPU
  - Thread Synchronization
  - Algorithm bottlenecks starve the GPU(s)
- Single GPU
  - Memory operations – blocking, serial, unnecessary
  - **Too much synchronization - device, context, stream, default stream, implicit**
  - CPU GPU Overlap – avoid excessive communication
- Multi GPU
  - Communication between GPUs
  - Lack of Stream Overlap in memory management, kernel execution

Too much Synchronization



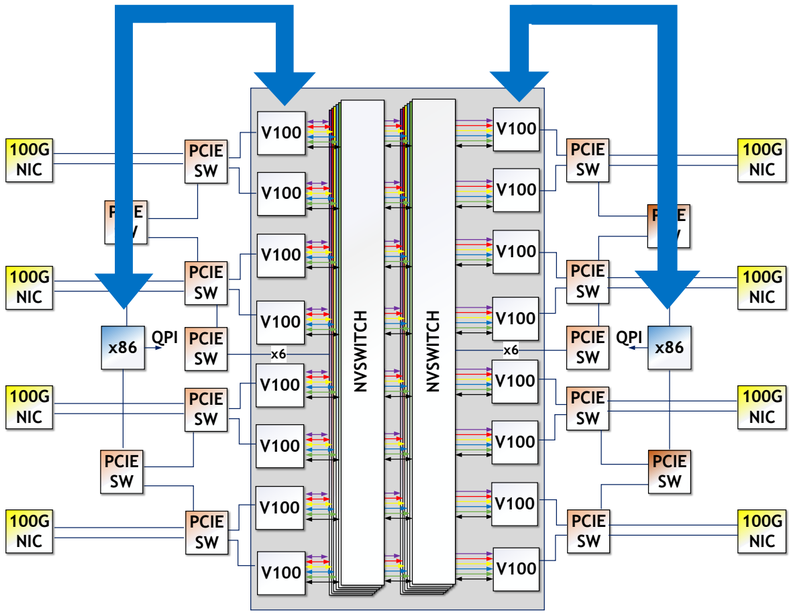

Synchronization Removed



# IMPROVING CPU LAUNCH OVERHEADS

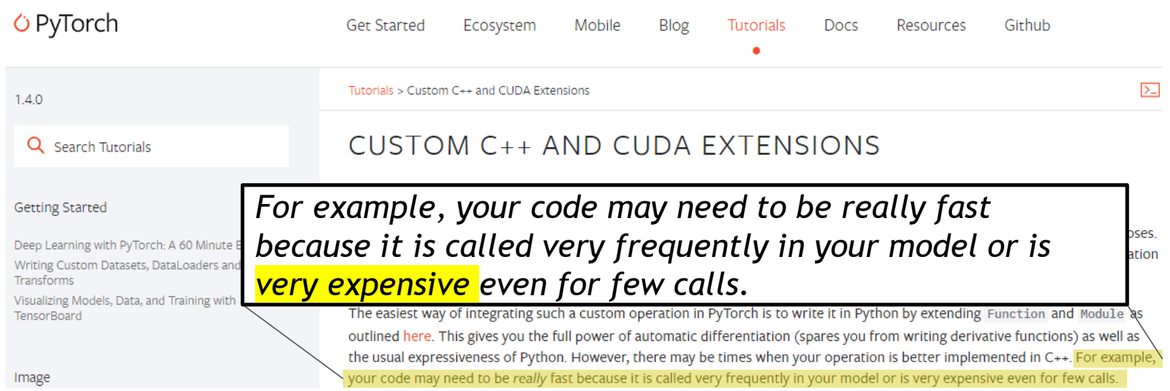
## Use NUMA-aware CPU affinity

Use CPU cores that are directly attached




DGX2 Topology

## Use framework features



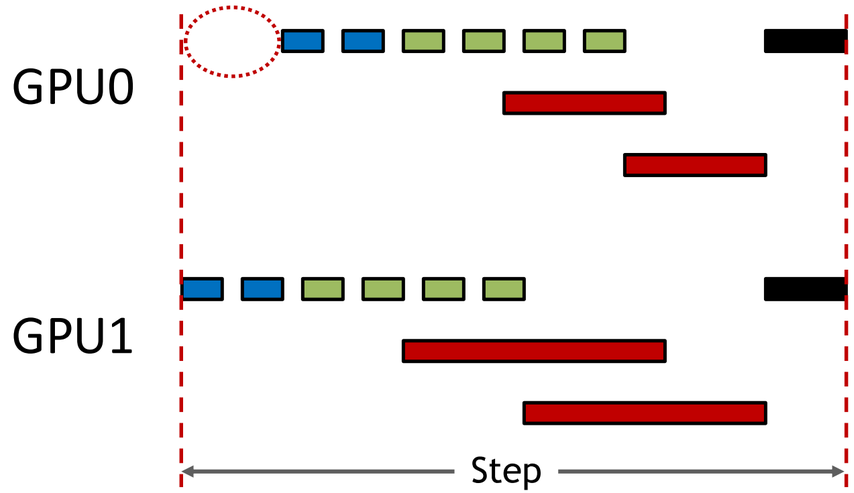

*For example, your code may need to be really fast because it is called very frequently in your model or is very expensive even for few calls.*

https://pytorch.org/tutorials/advanced/cpp_extension.html

# OTHER COMMON ISSUES TO WATCH OUT FOR (I)

Input pipeline stalls: batches fail to arrive on-time at the beginning of step

- Common in CNNs (sometimes perform extensive pre-processing e.g. image decode, augmentation)
- Manifest as large pre-step gaps in timeline/ analysis plots
- Some remedies:
  - NVIDIA Data Loading Library (DALI): https://developer.nvidia.com/DALI
  - Spawn multiple CPU processes for data-loading IO
  - Use pinned (page-locked) memory



# OTHER COMMON ISSUES TO WATCH OUT FOR (II)

Workload imbalance issues: variables-sized inputs lead to unexpected execution

- Examples:
  - Failure to pad batch dimensions to multiples of 8 to target Tensor cores (https://devblogs.nvidia.com/programming-tensor-cores-cuda-9/)
  - Images used in CNN training are not a constant size
- Debug procedure:
  - Annotate each step's NVTX range with batch size information and collect profiles
  - Use data-analytics to pinpoint steps that exhibit abnormally large forward & backward propagation times

# CONCLUSION

Home > Tools

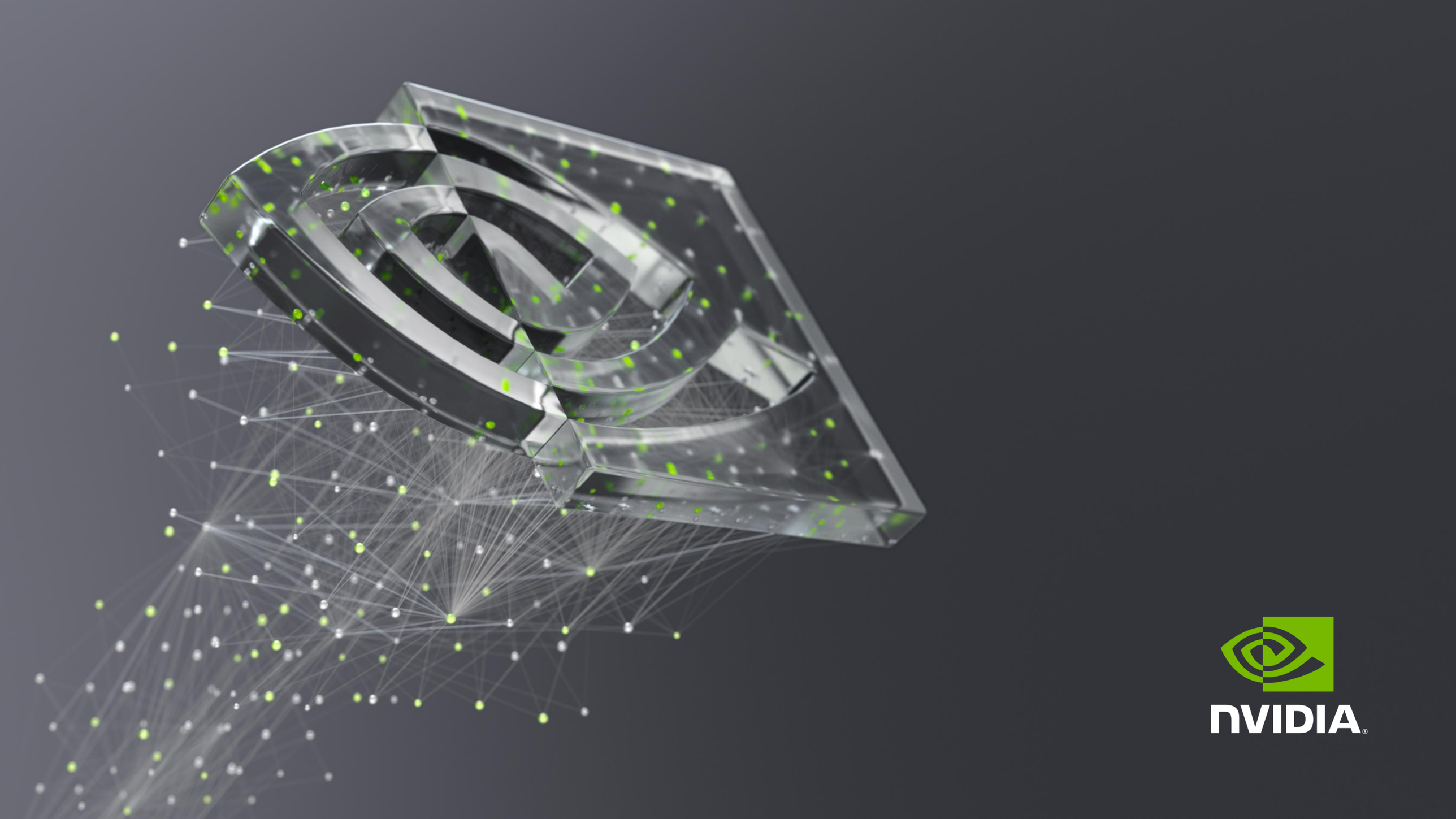
## NVIDIA Nsight Systems

**NVIDIA® Nsight™ Systems** is a system-wide performance analysis tool designed to visualize an application's algorithms, help you identify the largest opportunities to optimize, and tune to scale efficiently across any quantity or size of CPUs and GPUs; from large server to our smallest SoC.

Download Now

developer.nvidia.com/nsight-systems

- Nsight Systems + Python ➔ performance debug of Data Center scaled workloads
- Multi-node, multi-GPU workloads can be distilled into a traditional optimization effort
- Nsight Systems Exposes Many Common Causes of Jitter


NVIDIA