Efficient Deep Learning Systems
Optimizing training pipelines

Max Ryabinin

2026

Plan for today

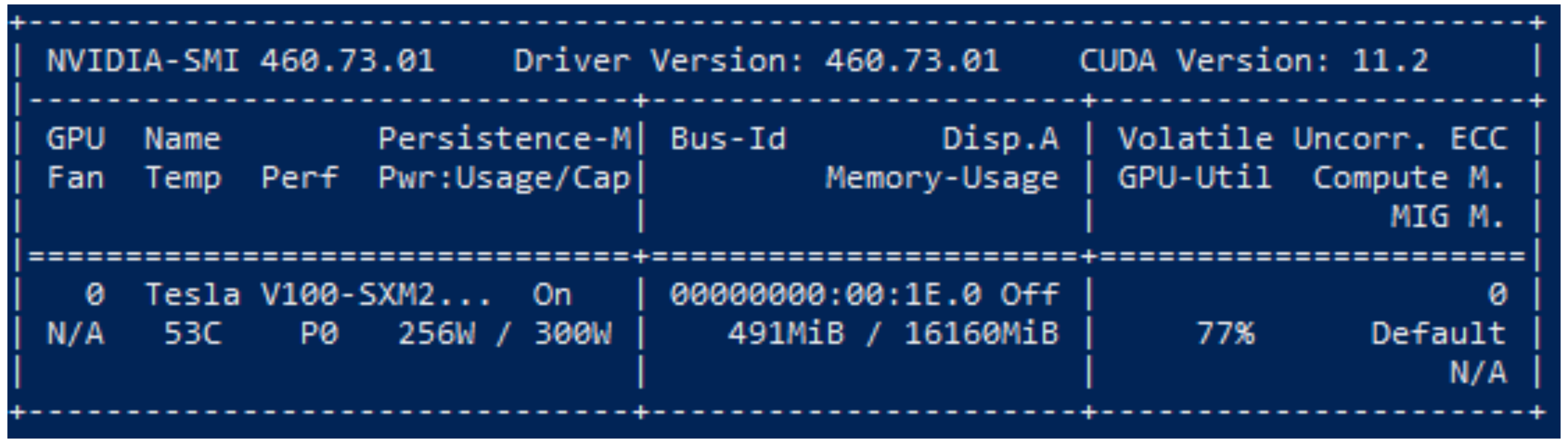
- Understanding performance limits
- Mixed precision training
    - When and why to use it
    - How to enable it and utilize it to the fullest
    - Dealing with stability in training
- Storing and loading training data efficiently
- Profiling DL code

DL performance indicators

- When do we want to optimize our code?
    - When do we know we’re “good enough”?
- Ultimately, **hardware performance** is the limiting factor
- If most of the time is spent on *useful* computations, then our code is close to optimal
- How do we know if we're close to the limit?

nvidia-smi

- On most Linux systems with a GPU driver, you have an easy way to check GPU status



```
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 460.73.01    Driver Version: 460.73.01    CUDA Version: 11.2     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  Tesla V100-SXM2...  On   | 00000000:00:1E.0 Off |                    0 |
| N/A   53C    P0   256W / 300W |    491MiB / 16160MiB |     77%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+
```

arthurchiao.art/blog/understanding-gpu-performance

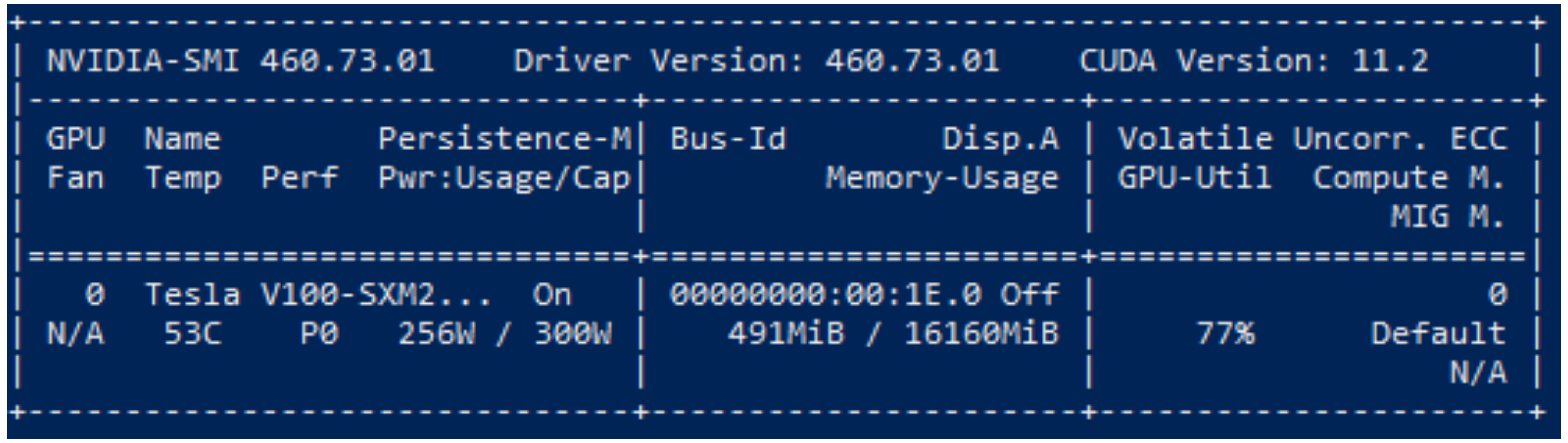
nvidia-smi (the wrong way)

- On most Linux systems with a GPU driver, you have an easy way to check GPU status

- However, its utilization metric **does not show what you think**!

- It shows the percentage of time when the GPU was running *anything*

  - A dummy kernel waiting for CPU-GPU sync could result in 100% “utilization”

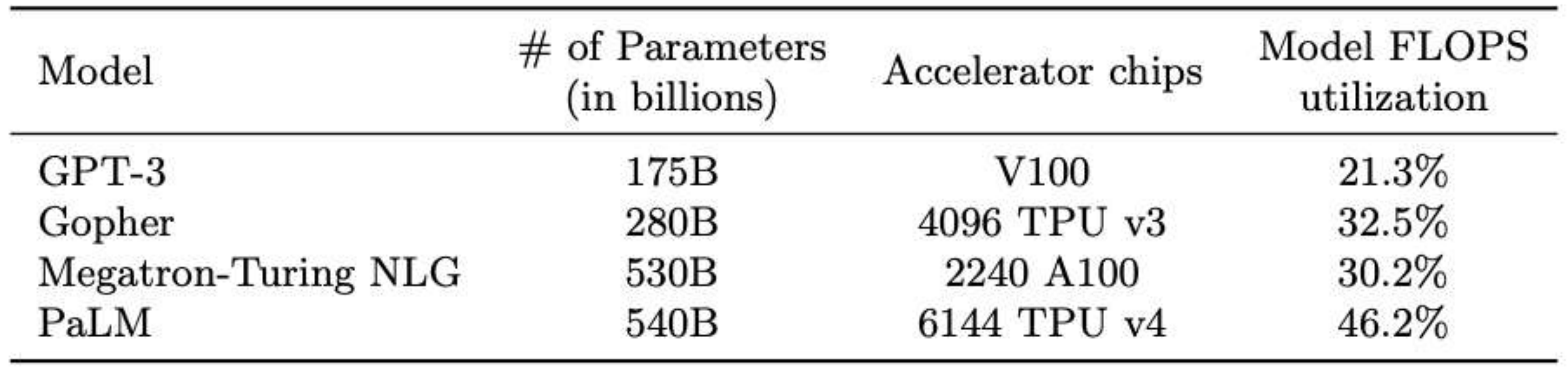


```
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 460.73.01    Driver Version: 460.73.01    CUDA Version: 11.2     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  Tesla V100-SXM2...  On   | 00000000:00:1E.0 Off |                    0 |
| N/A   53C    P0   256W / 300W |    491MiB / 16160MiB |     77%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+
```

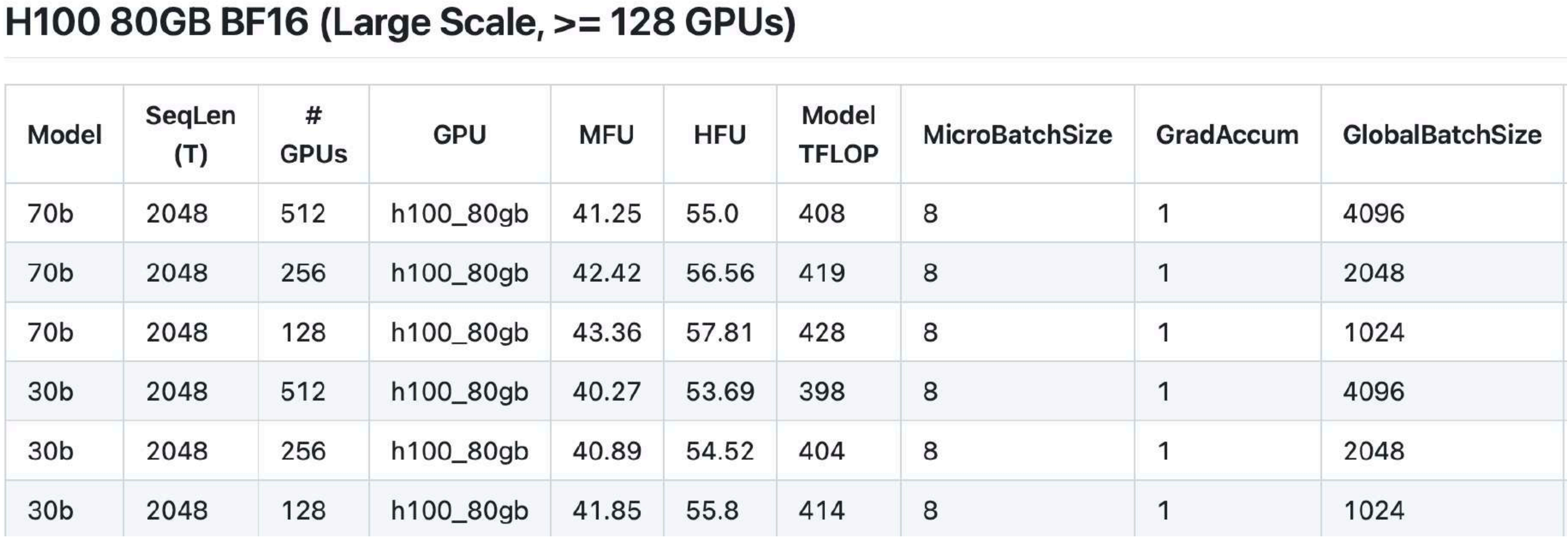
arthurchiao.art/blog/understanding-gpu-performance

The right way: Model FLOPS Utilization

- Defined as the ratio of observed FLOPS (floating point operations/second) to theoretical maximum FLOPS on given hardware [1]
- A system-independent metric that indicates end-to-end performance
- Rule of thumb: MFU >45% is a good efficiency target
- Many benchmarks published [2], but beware of different ways to compute FLOPS!



Model	# of Parameters (in billions)	Accelerator chips	Model FLOPS utilization
GPT-3	175B	V100	21.3%
Gopher	280B	4096 TPU v3	32.5%
Megatron-Turing NLG	530B	2240 A100	30.2%
PaLM	540B	6144 TPU v4	46.2%



H100 80GB BF16 (Large Scale, >= 128 GPUs)

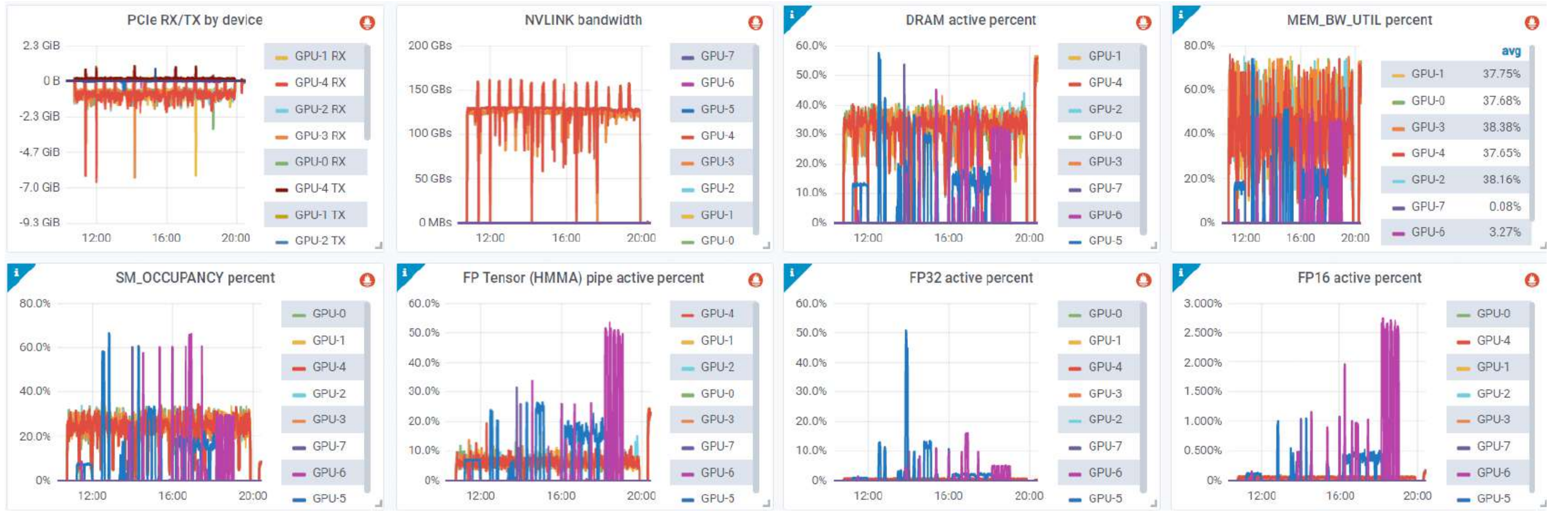
Model	SeqLen (T)	# GPUs	GPU	MFU	HFU	Model TFLOP	MicroBatchSize	GradAccum	GlobalBatchSize
70b	2048	512	h100_80gb	41.25	55.0	408	8	1	4096
70b	2048	256	h100_80gb	42.42	56.56	419	8	1	2048
70b	2048	128	h100_80gb	43.36	57.81	428	8	1	1024
30b	2048	512	h100_80gb	40.27	53.69	398	8	1	4096
30b	2048	256	h100_80gb	40.89	54.52	404	8	1	2048
30b	2048	128	h100_80gb	41.85	55.8	414	8	1	1024

[1] PaLM: Scaling Language Modeling with Pathways. Chowdhery et al., 2022

[2] github.com/mosaicml/llm-foundry/tree/main/scripts/train/benchmarking#results

The right*est* way: ~~MFU~~ it depends :)

- MFU assumes a fixed compute type for everything
- Sometimes FLOPs might be input/operation-dependent
- We might not account for operations beside forward/backward/step
  - HFU (Hardware FLOPs Utilization) could be a solution
- If you're interested in the system performance, tokens/second could do
- Various indicators from DCGM report actual use of full GPU resources



GPU compute saturation: takeaways

- The performance limits are defined by how much of hardware we can effectively utilize

- Use MFU/HFU as a first approximation

    - Or hardware counters from DCGM [1] for most accurate measures

- Beware of different ways to compute FLOPs (both for model computations and for hardware)

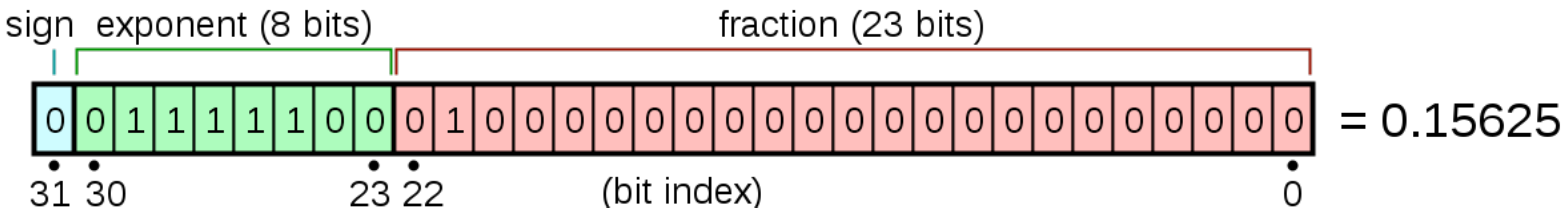
[1] docs.nvidia.com/datacenter/dcgm/latest/user-guide/feature-overview.html#profiling-metrics

Floating point numbers

- Neural networks require real numbers…

- …which need to be represented in finite memory

- Single precision (FP32) is the default format with 4 bytes of storage



$$\textbf{value} = (-1)^{\textbf{sign}} \times 2^{\textbf{E}-127} \times \left(1 + \sum_{i=1}^{23} b_{23-i} 2^{-i}\right)$$

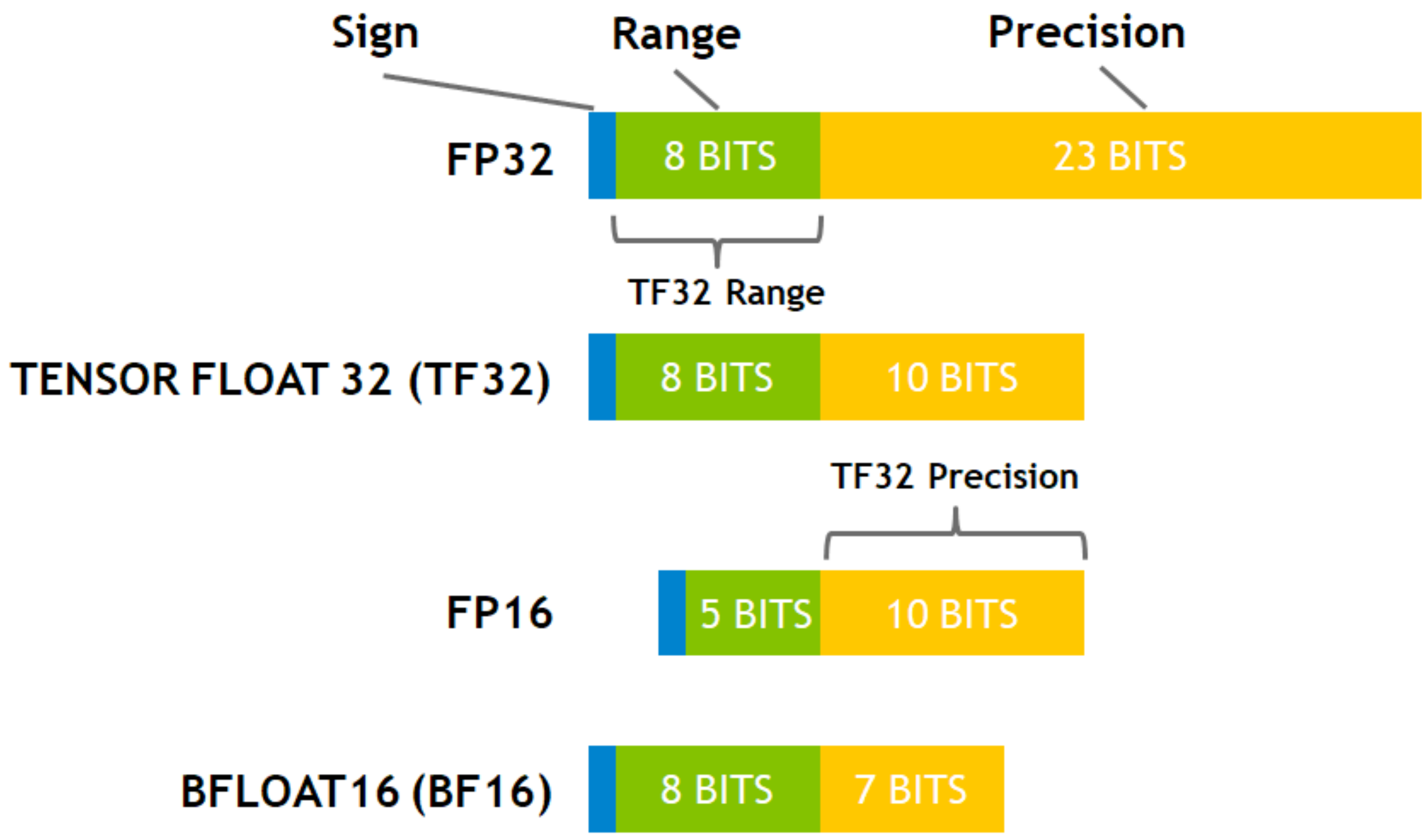
- Special values (0, NaN, ±inf) are encoded by exponent values

Why use low precision?

- Can we go smaller than 32 bits? Should we?
- Key benefits:
  - Reduced memory usage (duh)
  - Faster performance (due to higher arithmetic intensity or smaller communication footprint)
  - Can use specialized hardware for even faster computation
- Makes your code prone to spectacular explosions :)

Floating point formats

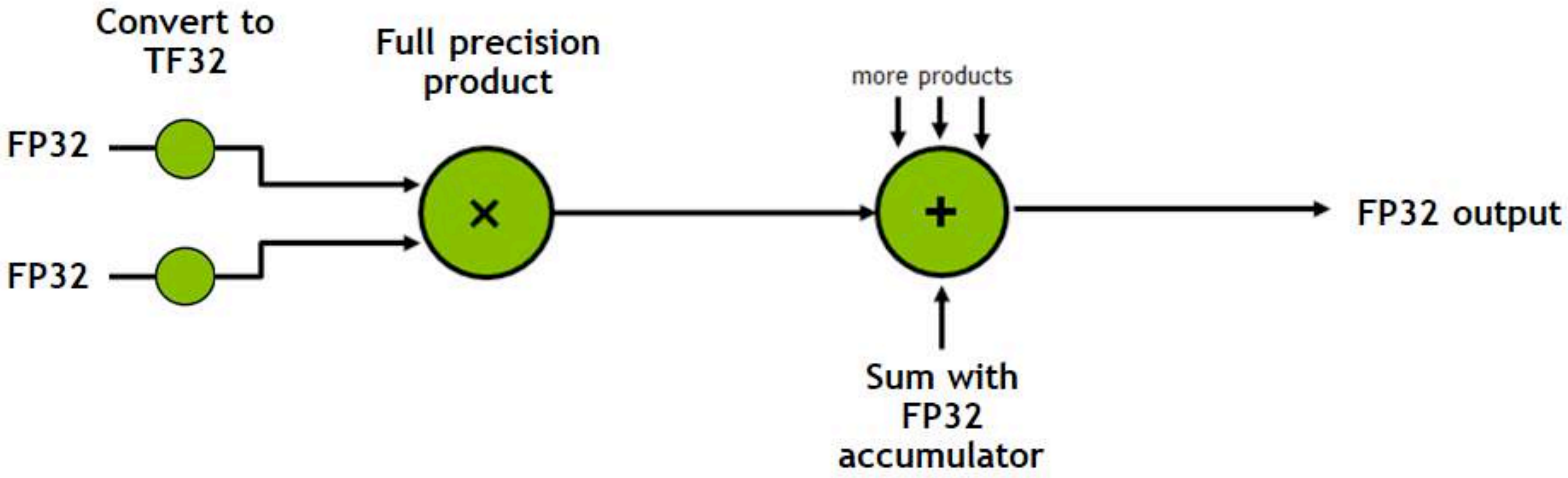
- Naive FP16 is not the only option!
- Specialized formats preserve dynamic range for computations



src: developer.nvidia.com/blog/accelerating-ai-training-with-tf32-tensor-cores

Switching to lower precision

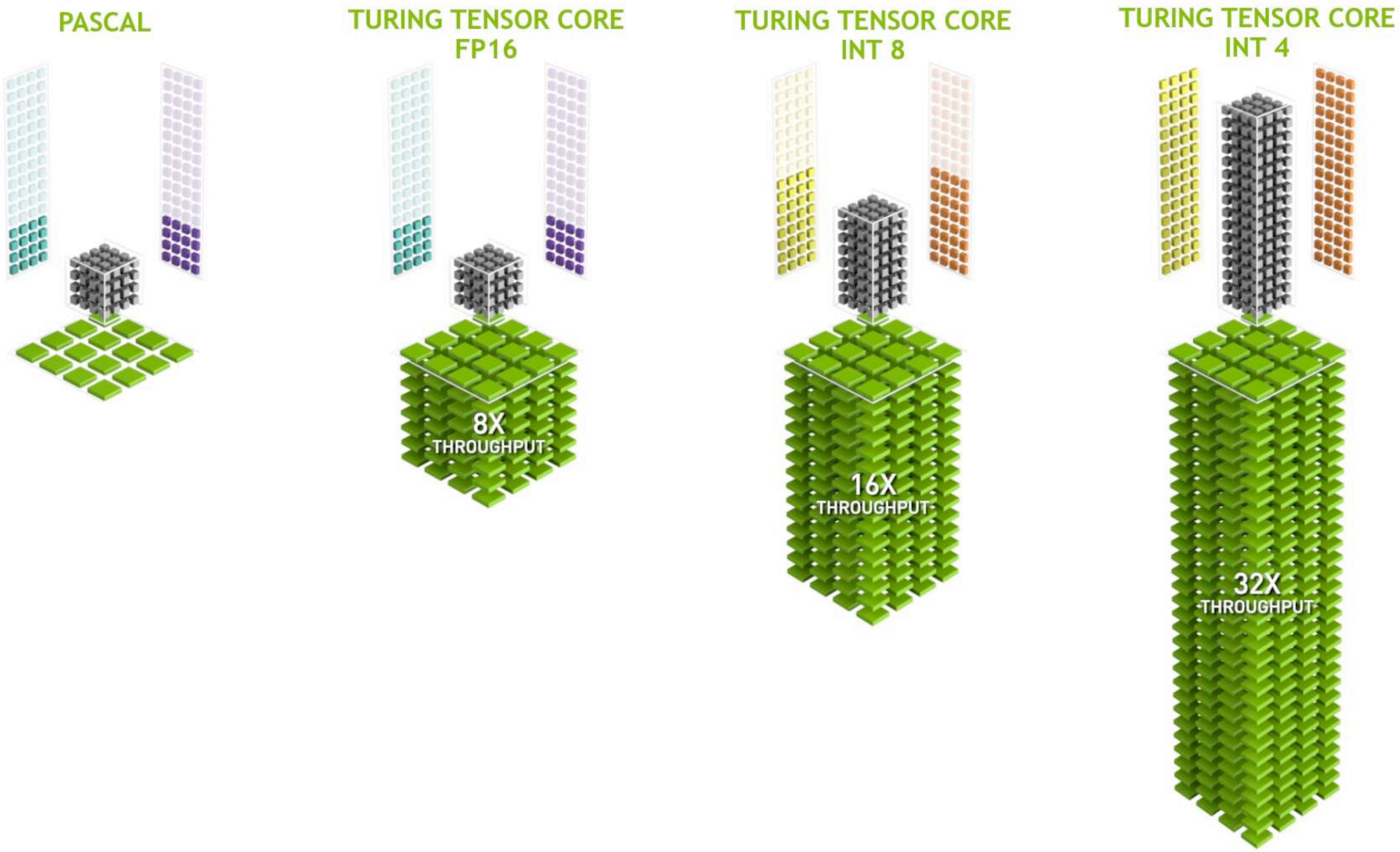
- FP16 exists since CUDA 8, just allocate the tensor/cast it to `half`
- BF16 is available on CPUs, TPUs and recent GPUs [1], `Tensor.bfloat16()` in PyTorch
- TF32 can be enabled for you on Ampere GPUs
  (was enabled in PyTorch by default until 1.12)
    - Never exposed as a data type, only as a type for specific operations [2]



[1] pytorch.org/xla/release/1.9/index.html#xla-tensors-and-bfloat16
[2] developer.nvidia.com/blog/accelerating-ai-training-with-tf32-tensor-cores

Tensor Cores

- Specialized computation units available in latest generations of NVIDIA GPUs (since Volta)
- Allow the user to speed up $D = A \times B + C$ by up to 8-16x (claimed)



nvlabs.github.io/eccv2020-mixed-precision-tutorial/files/dusan_stosic-training-neural-networks-with-tensor-cores.pdf

Tensor Cores

- Specialized computation units available in latest generations of NVIDIA GPUs (since Volta)
- Allow the user to speed up $D = A \times B + C$ by up to 8-16x (claimed)
- Enabled not only for TF32/FP16/BF16 (Ampere), but even for INT8/INT4
- You do not specify their usage manually!

nvlabs.github.io/eccv2020-mixed-precision-tutorial/files/dusan_stosic-training-neural-networks-with-tensor-cores.pdf

Utilizing Tensor Cores

- To enable them, you either need recent CUDA or specific size constraints:

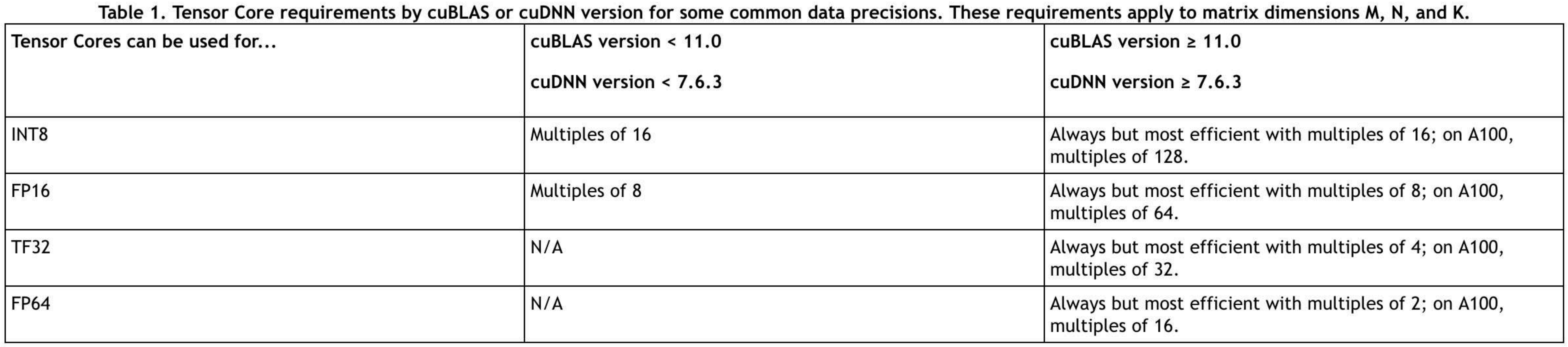


Table 1. Tensor Core requirements by cuBLAS or cuDNN version for some common data precisions. These requirements apply to matrix dimensions M, N, and K.

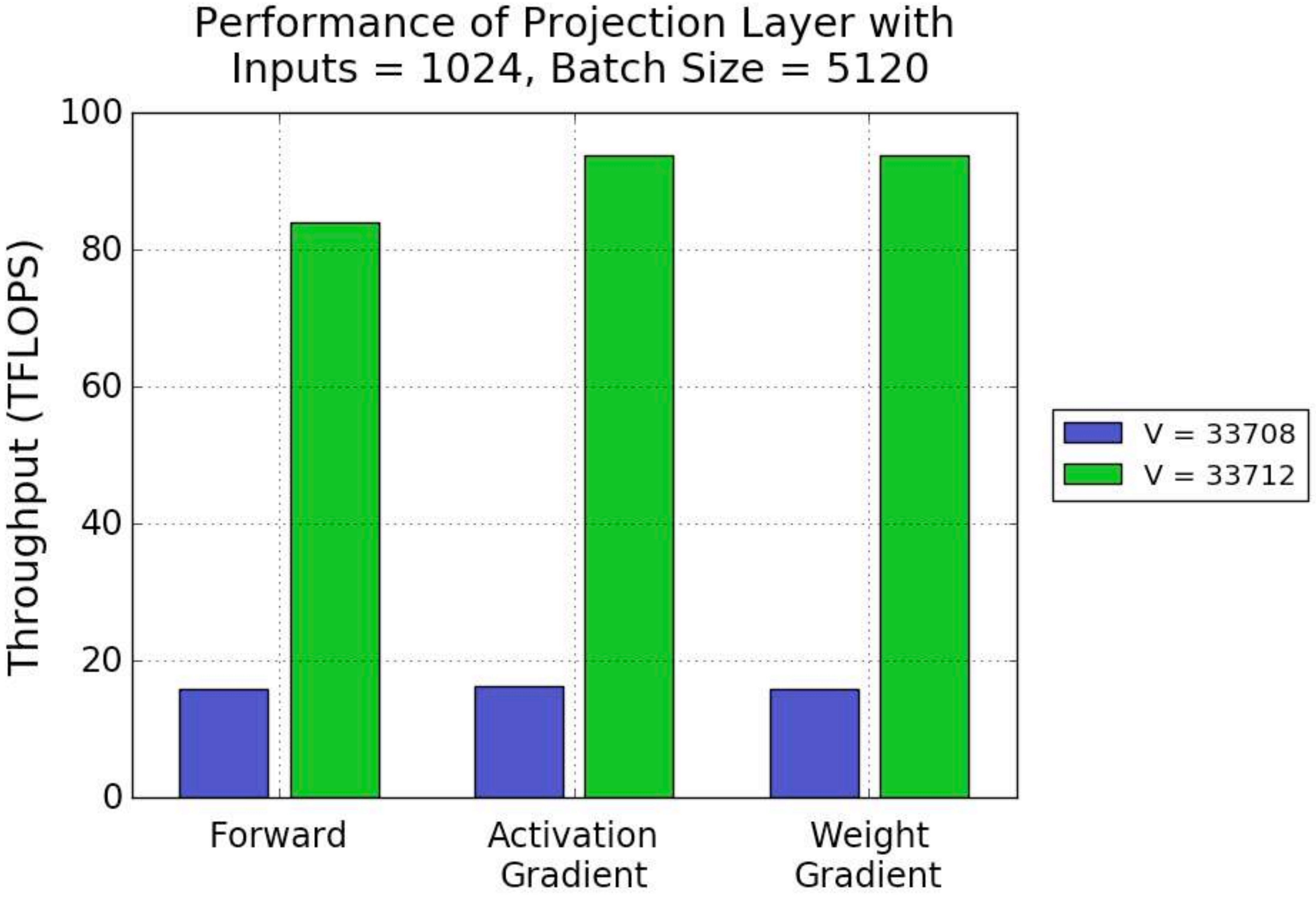
Tensor Cores can be used for...	cuBLAS version < 11.0 cuDNN version < 7.6.3	cuBLAS version ≥ 11.0 cuDNN version ≥ 7.6.3
INT8	Multiples of 16	Always but most efficient with multiples of 16; on A100, multiples of 128.
FP16	Multiples of 8	Always but most efficient with multiples of 8; on A100, multiples of 64.
TF32	N/A	Always but most efficient with multiples of 4; on A100, multiples of 32.
FP64	N/A	Always but most efficient with multiples of 2; on A100, multiples of 16.

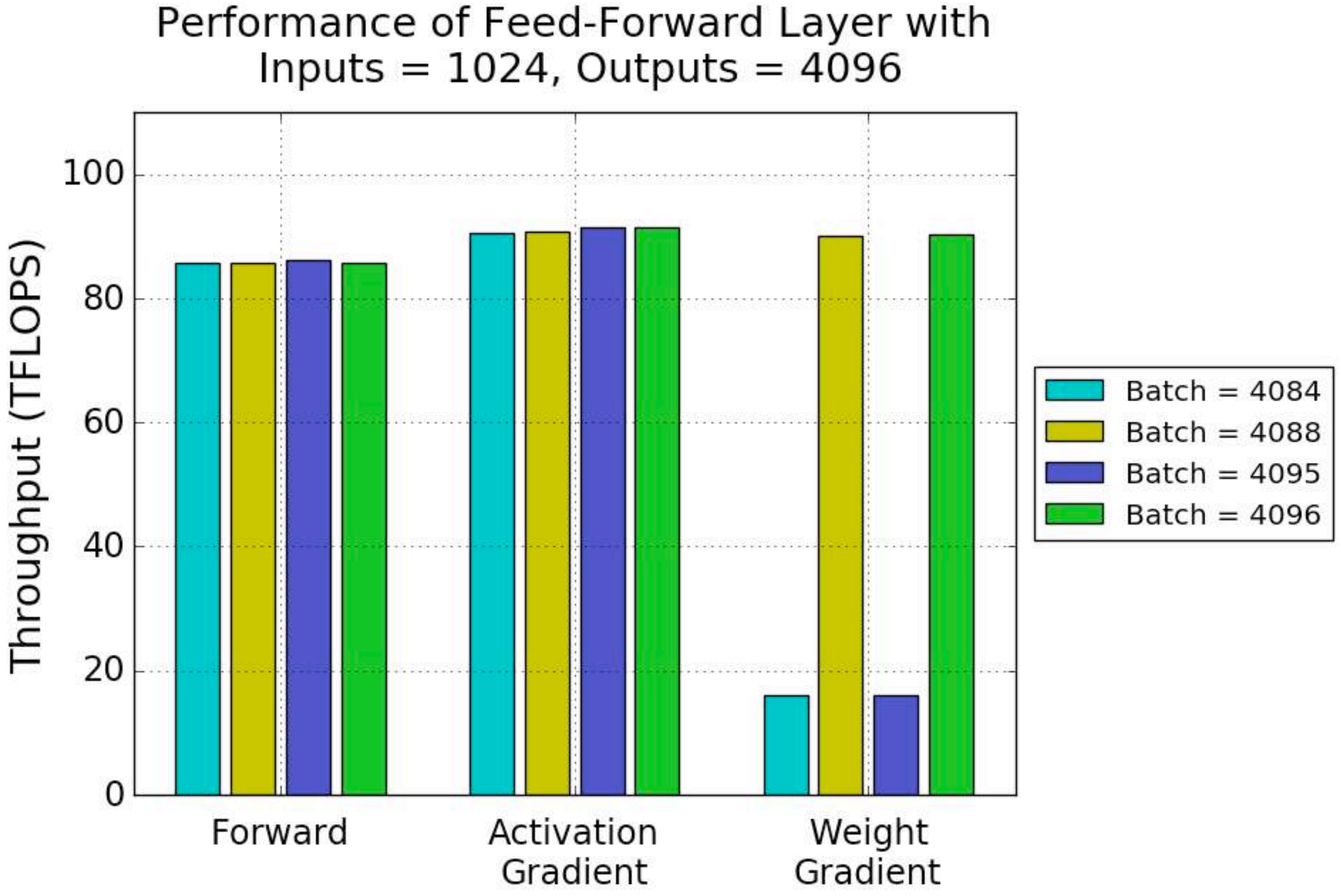
[1] docs.nvidia.com/deeplearning/performance/dl-performance-matrix-multiplication/index.html#requirements-tc

[2] developer.download.nvidia.com/video/gputechconf/gtc/2019/presentation/s9926-tensor-core-performance-the-ultimate-guide.pdf

Utilizing Tensor Cores

- To enable them, you either need recent CUDA or specific size constraints:





[1] docs.nvidia.com/deeplearning/performance/dl-performance-matrix-multiplication/index.html#requirements-tc

[2] developer.download.nvidia.com/video/gputechconf/gtc/2019/presentation/s9926-tensor-core-performance-the-ultimate-guide.pdf

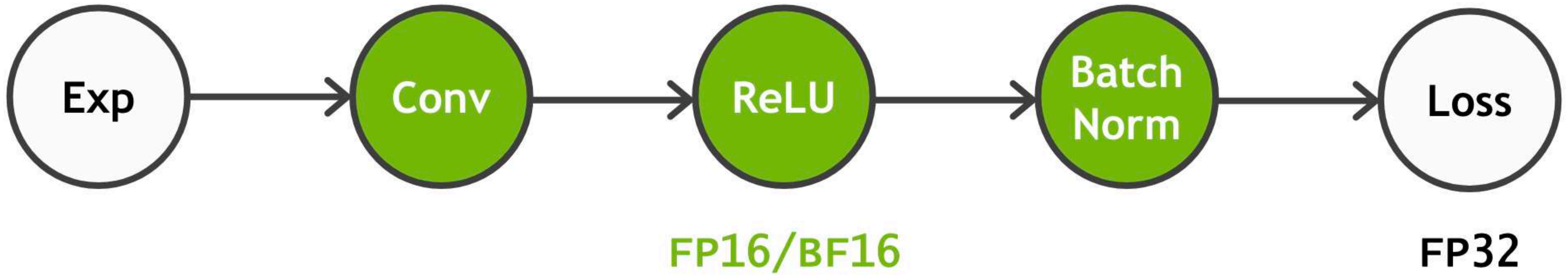
Utilizing Tensor Cores

- To enable them, you either need recent CUDA or specific size constraints:
- Run GPU profiler to check if they are used ([i|s|h](\d)+ in kernel names)
- Also, DL profilers can indicate Tensor Core eligibility and usage

[1] docs.nvidia.com/deeplearning/performance/dl-performance-matrix-multiplication/index.html#requirements-tc
[2] developer.download.nvidia.com/video/gputechconf/gtc/2019/presentation/s9926-tensor-core-performance-the-ultimate-guide.pdf

Mixed precision training

- Training in pure FP16 hardly works
- Some operations (e.g. matrix multiplication) can work, others (softmax, batch normalization) need higher precision
- Mixed precision training casts layer activations to appropriate data types
- Supported in popular DL frameworks (e.g. torch.cuda.amp)



- Increases the training throughput due to the use of Tensor Cores (MFU trickier to compute)
- Decreases the memory usage by half… or not?

Memory savings of AMP

- Let’s count the number of bytes per parameter for standard training with Adam:

FP32:

- Parameters — 4 bytes
- Gradients — 4 bytes
- Optimizer statistics — 8 bytes

16 bytes per parameter in total

AMP:

- Parameters — 2 bytes
- Master parameters — 4 bytes
- Gradients — 2 bytes (sometimes 4)
- Optimizer statistics — 8 bytes

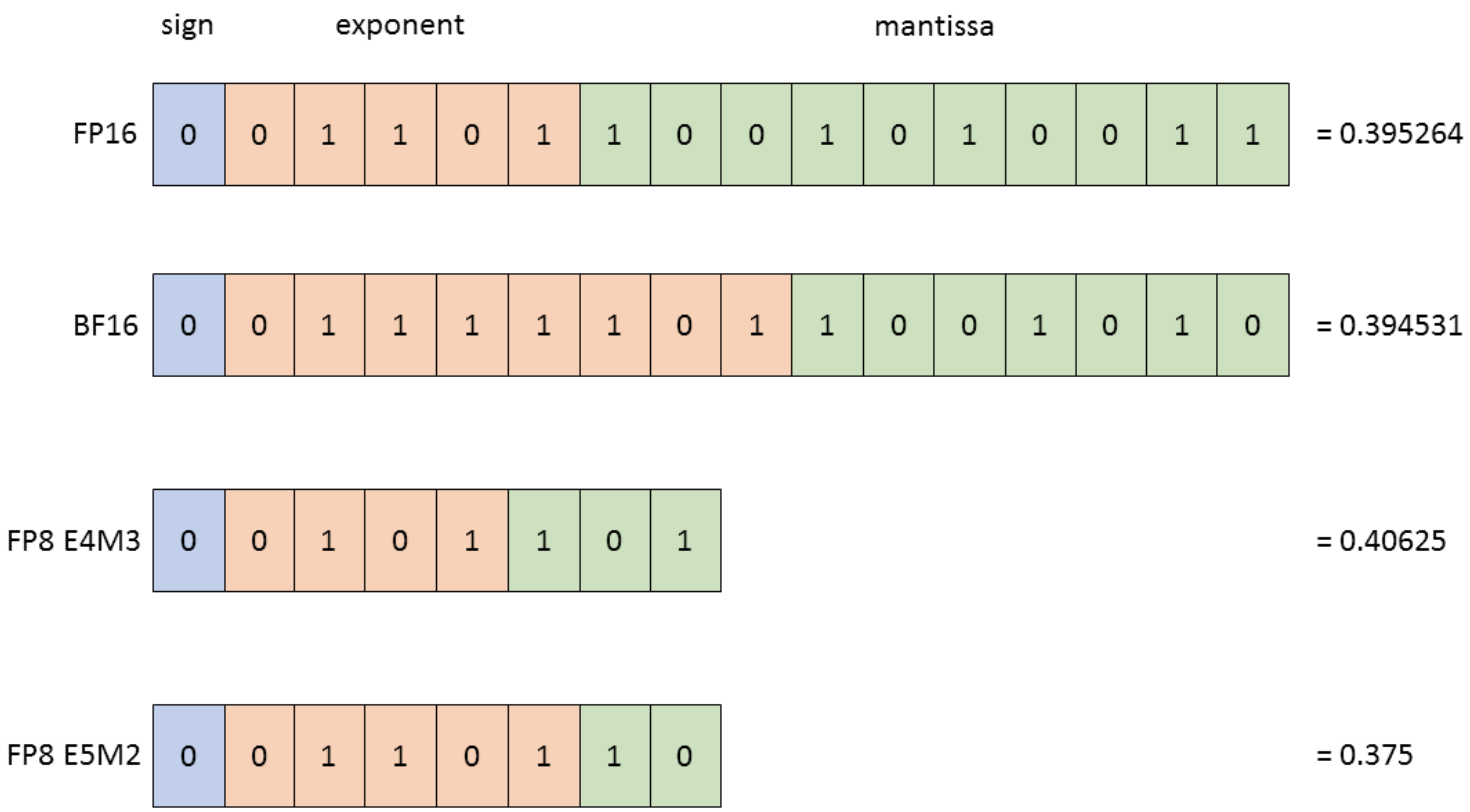
Also **16 bytes per parameter**!

- The only major savings come from reduced activation memory

FP8 training

- On latest hardware (e.g., H100), we have even lower precision formats

- E4M3 is used for weights and activations, E5M2 is best for gradients

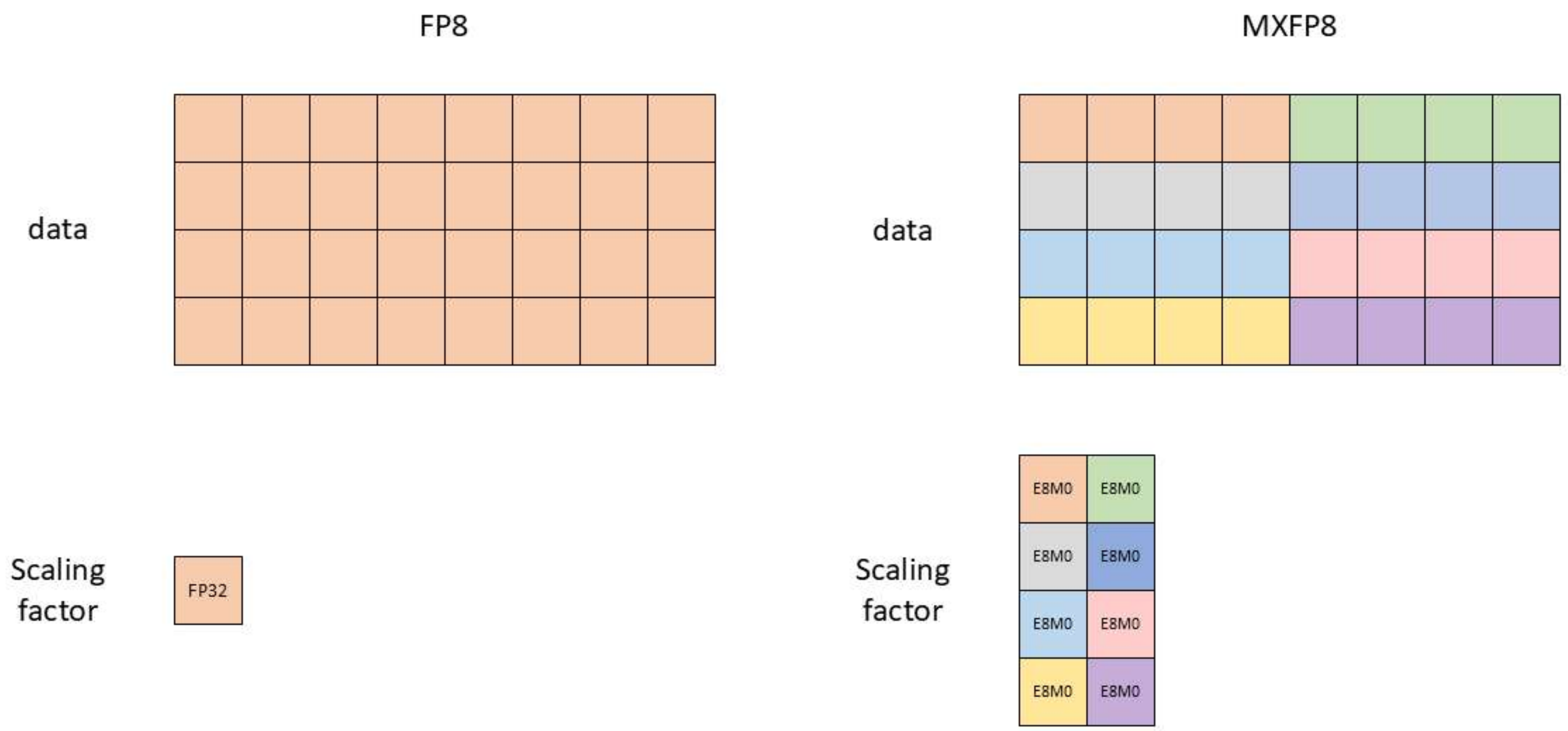


	sign			exponent								mantissa					
FP16	0	0	1	1	0	1	1	0	0	1	0	1	0	0	1	1	= 0.395264
BF16	0	0	1	1	1	1	1	0	1	1	0	0	1	0	1	0	= 0.394531
FP8 E4M3	0	0	1	0	1	1	0	1									= 0.40625
FP8 E5M2	0	0	1	1	0	1	1	0									= 0.375

FP8 Formats for Deep Learning. Micikevicius et al., 2022

FP8 training

- On latest hardware (e.g., H100), we have even lower precision formats
- E4M3 is used for weights and activations, E5M2 is best for gradients
- Extra tricks, e.g. per-tensor or per-block (MX) scaling, required to maintain accuracy
- In PyTorch: github.com/pytorch/ao/tree/main/torchao/float8
- Also, github.com/NVIDIA/TransformerEngine can leverage this

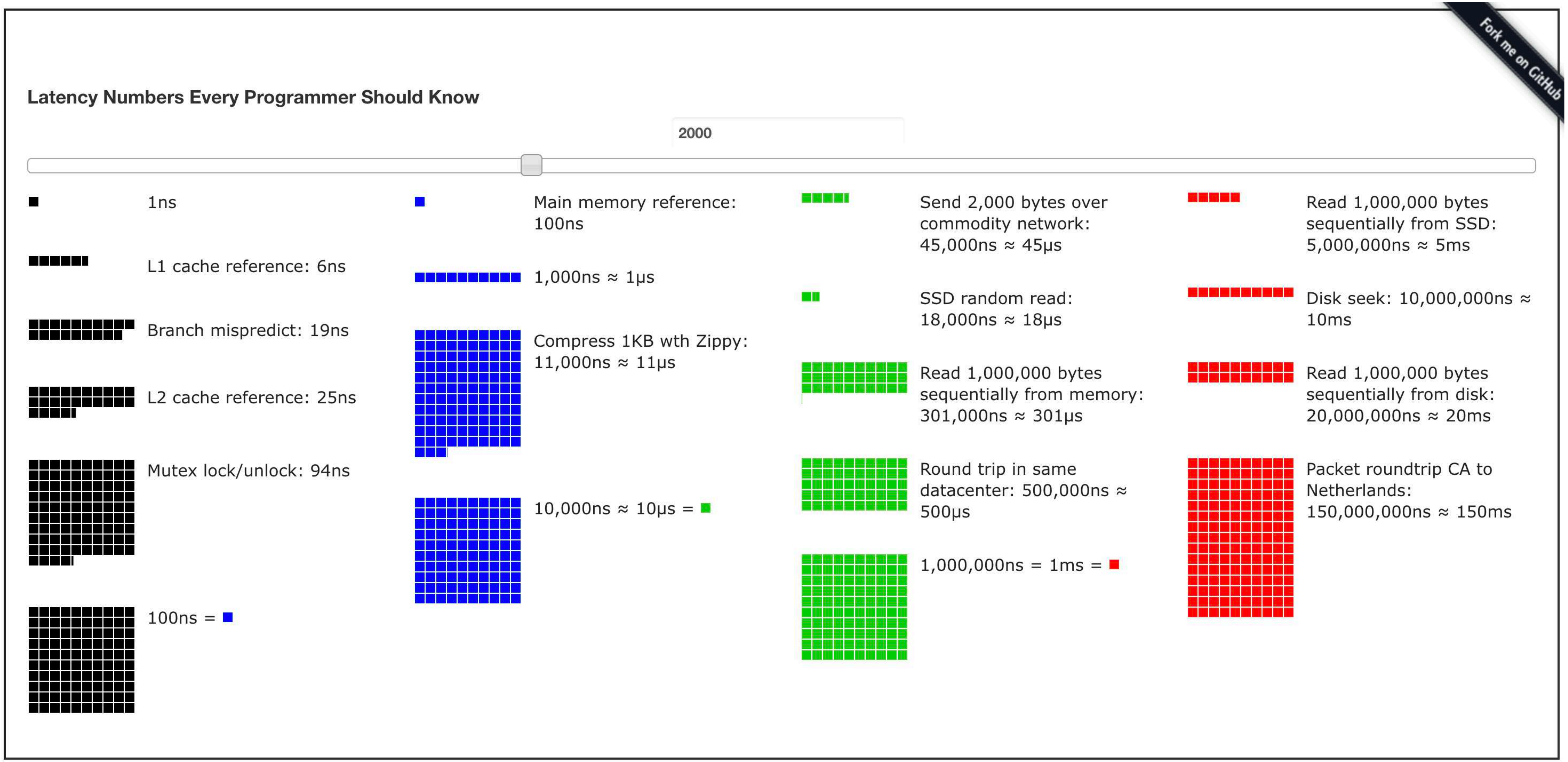


FP8 Formats for Deep Learning. Micikevicius et al., 2022

AMP: takeaways

- Use more efficient data types when available
- Mind the sizes/operation types to preserve accuracy
- Don’t expect significant memory savings for large models
- In many cases, this is easy to integrate through standard tools

Bottlenecks in data loading



Latency Numbers Every Programmer Should Know

2000

1ns

L1 cache reference: 6ns

Branch mispredict: 19ns

L2 cache reference: 25ns

Mutex lock/unlock: 94ns

100ns = 🟦

Main memory reference: 100ns

1,000ns ≈ 1µs

Compress 1KB wth Zippy: 11,000ns ≈ 11µs

10,000ns ≈ 10µs = 🟩

Send 2,000 bytes over commodity network: 45,000ns ≈ 45µs

SSD random read: 18,000ns ≈ 18µs

Read 1,000,000 bytes sequentially from memory: 301,000ns ≈ 301µs

Round trip in same datacenter: 500,000ns ≈ 500µs

1,000,000ns = 1ms = 🟥

Read 1,000,000 bytes sequentially from SSD: 5,000,000ns ≈ 5ms

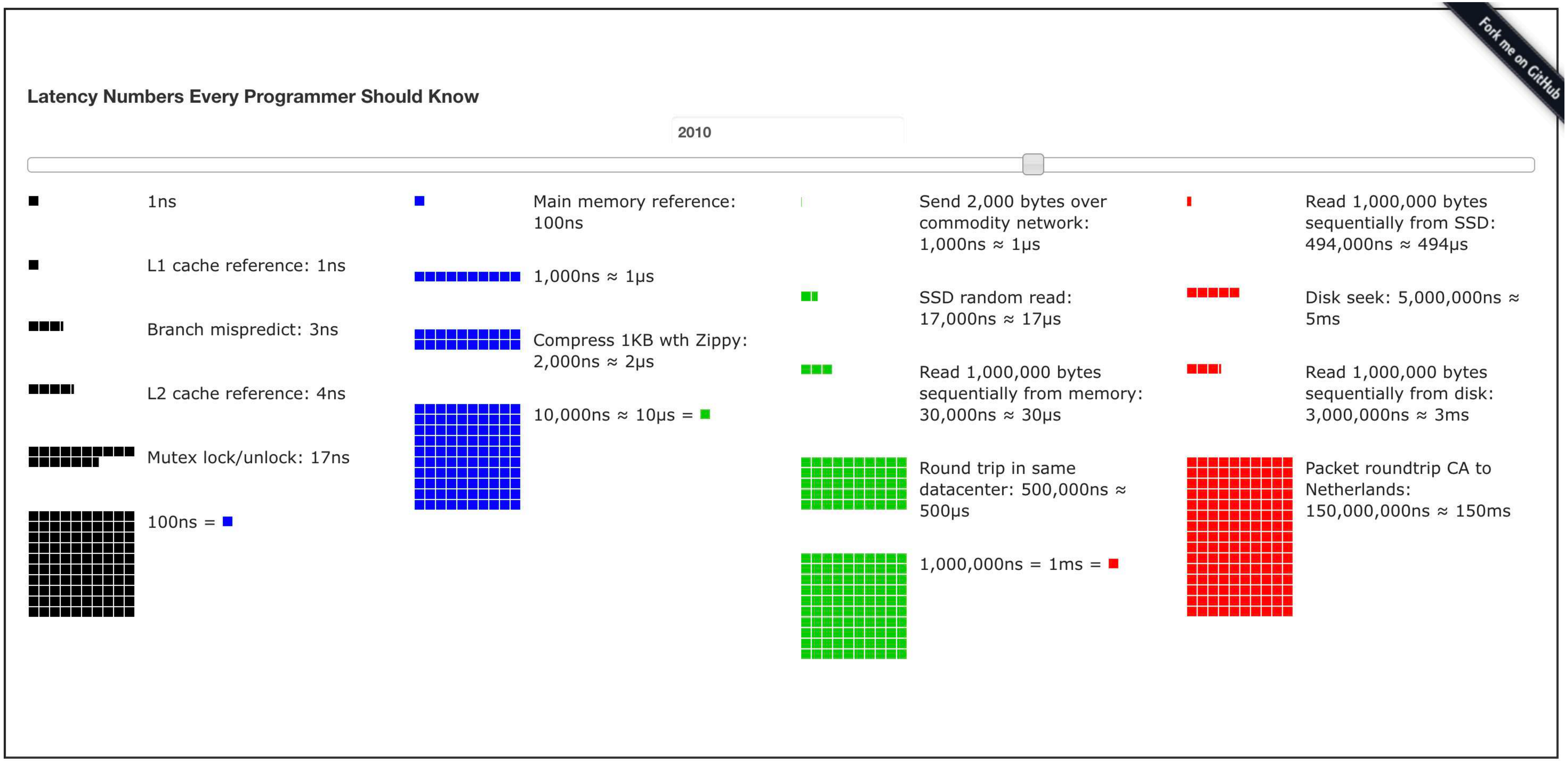
Disk seek: 10,000,000ns ≈ 10ms

Read 1,000,000 bytes sequentially from disk: 20,000,000ns ≈ 20ms

Packet roundtrip CA to Netherlands: 150,000,000ns ≈ 150ms

colin-scott.github.io/personal_website/research/interactive_latency.html

Bottlenecks in data loading



Latency Numbers Every Programmer Should Know

2010

1ns

L1 cache reference: 1ns

Branch mispredict: 3ns

L2 cache reference: 4ns

Mutex lock/unlock: 17ns

100ns = 🟦

Main memory reference: 100ns

1,000ns ≈ 1μs

Compress 1KB wth Zippy: 2,000ns ≈ 2μs

10,000ns ≈ 10μs = 🟩

Send 2,000 bytes over commodity network: 1,000ns ≈ 1μs

SSD random read: 17,000ns ≈ 17μs

Read 1,000,000 bytes sequentially from memory: 30,000ns ≈ 30μs

Round trip in same datacenter: 500,000ns ≈ 500μs

1,000,000ns = 1ms = 🟥

Read 1,000,000 bytes sequentially from SSD: 494,000ns ≈ 494μs

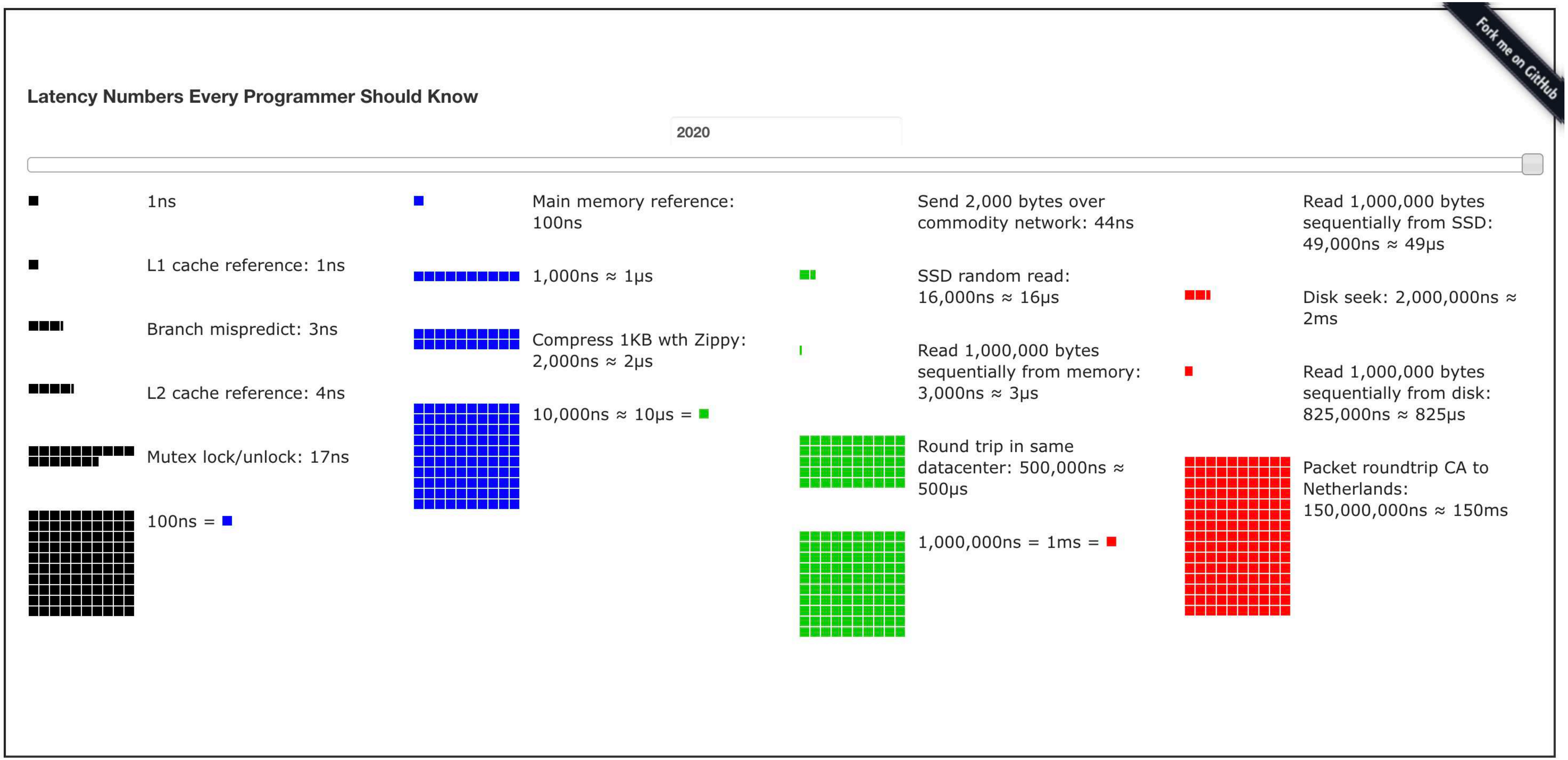
Disk seek: 5,000,000ns ≈ 5ms

Read 1,000,000 bytes sequentially from disk: 3,000,000ns ≈ 3ms

Packet roundtrip CA to Netherlands: 150,000,000ns ≈ 150ms

colin-scott.github.io/personal_website/research/interactive_latency.html

Bottlenecks in data loading



Latency Numbers Every Programmer Should Know

2020

1ns

L1 cache reference: 1ns

Branch mispredict: 3ns

L2 cache reference: 4ns

Mutex lock/unlock: 17ns

100ns = 🟦

Main memory reference: 100ns

1,000ns ≈ 1µs

Compress 1KB wth Zippy: 2,000ns ≈ 2µs

10,000ns ≈ 10µs = 🟩

Send 2,000 bytes over commodity network: 44ns

SSD random read: 16,000ns ≈ 16µs

Read 1,000,000 bytes sequentially from memory: 3,000ns ≈ 3µs

Round trip in same datacenter: 500,000ns ≈ 500µs

1,000,000ns = 1ms = 🟥

Read 1,000,000 bytes sequentially from SSD: 49,000ns ≈ 49µs

Disk seek: 2,000,000ns ≈ 2ms

Read 1,000,000 bytes sequentially from disk: 825,000ns ≈ 825µs

Packet roundtrip CA to Netherlands: 150,000,000ns ≈ 150ms

colin-scott.github.io/personal_website/research/interactive_latency.html

Bottlenecks in data loading

- Sometimes the models aren’t so compute-intensive…
- We still want to process the data efficiently!
- Need to be mindful of hardware/network performance and the CPU code
- Two components: <u>what to read</u> and <u>how to read</u>
- Obvious part: read data in parallel
  (several processes, asynchronously with computation)

Storage formats

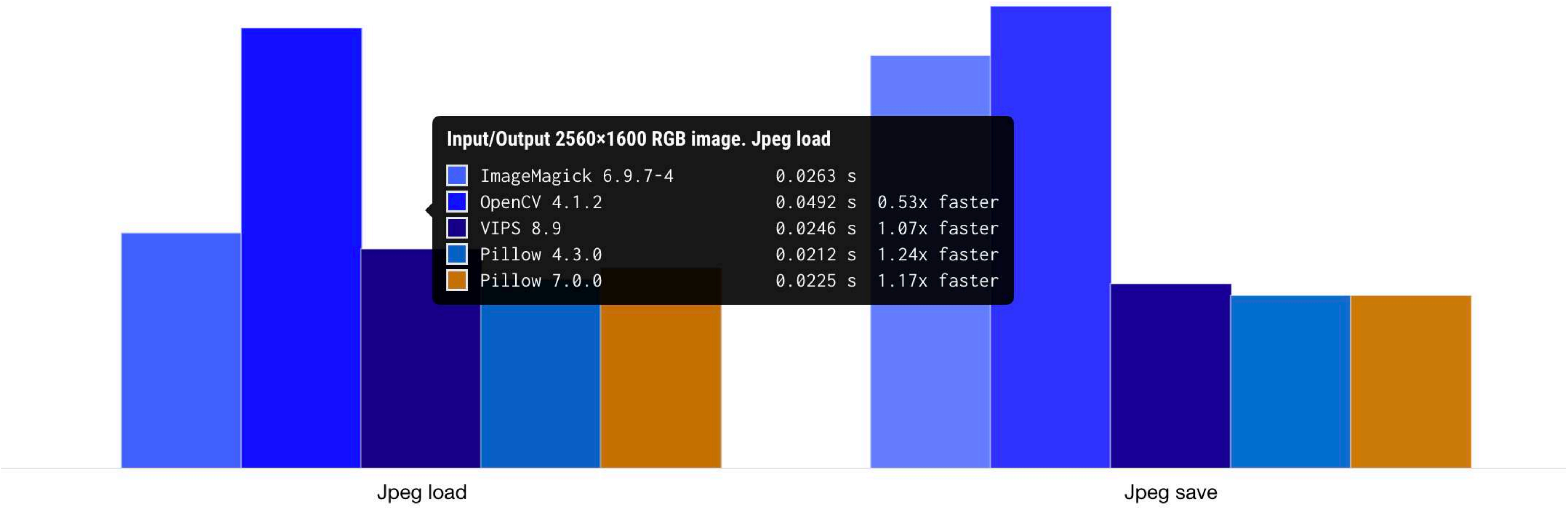
- Raw files are often easy to visualize, but storage-inefficient (especially when accessing external storage)
- In some cases, you might benefit from better formats:
  - For structured data, Apache Arrow/Protobuf/msgpack etc.
  - For images, apply non-random “heavy” processing before training
  - For language data, tokenize the texts and store integer indices only

Minimizing preprocessing time

- Reading the data and feeding it into the model can also be slow
    - For large images, you can be bound by CPU operations
    - For sequence data, you can waste time on padding tokens

Performance of image loading

- When reading images, consider the code that reads them :)
  - Default PIL.Image.Open can be highly inefficient!
    Use at least Pillow-SIMD
  - Use better decoders (e.g. jpegturbo, nvJPEG from DALI)



Input/Output 2560×1600 RGB image. Jpeg load

ImageMagick 6.9.7-4	0.0263 s	
OpenCV 4.1.2	0.0492 s	0.53x faster
VIPS 8.9	0.0246 s	1.07x faster
Pillow 4.3.0	0.0212 s	1.24x faster
Pillow 7.0.0	0.0225 s	1.17x faster

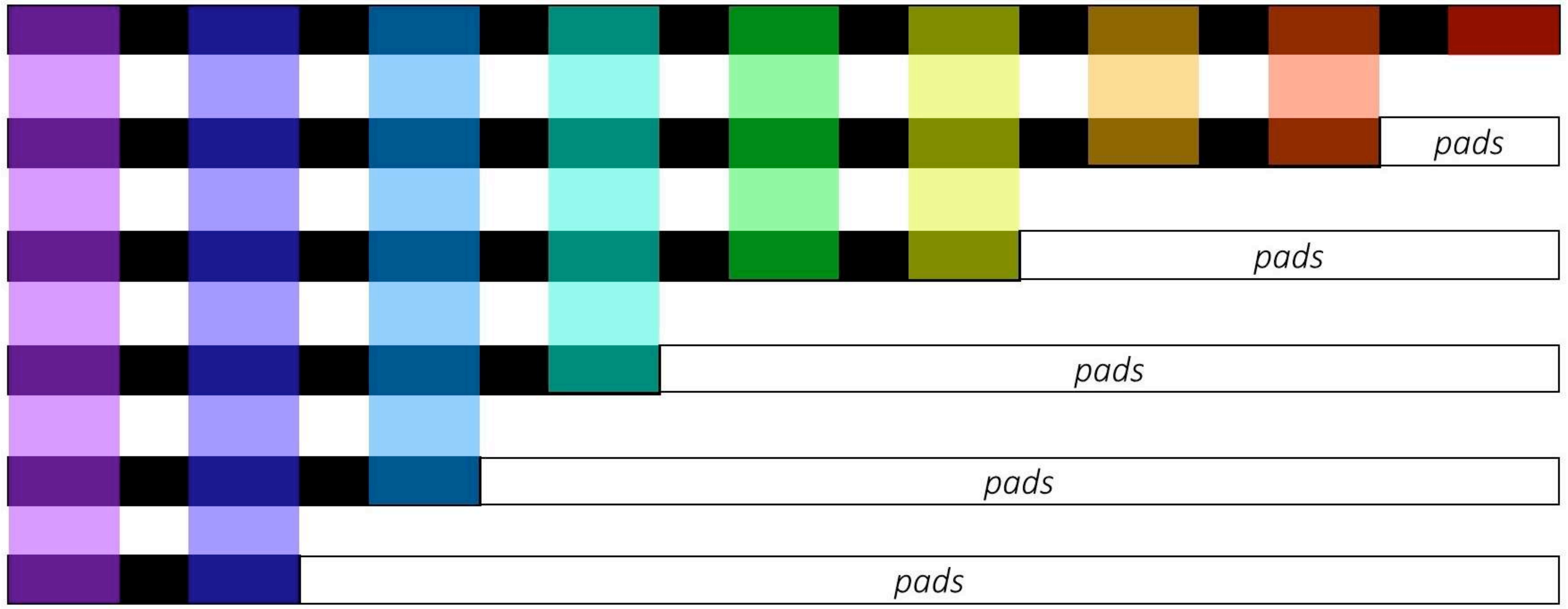
Performance of image loading

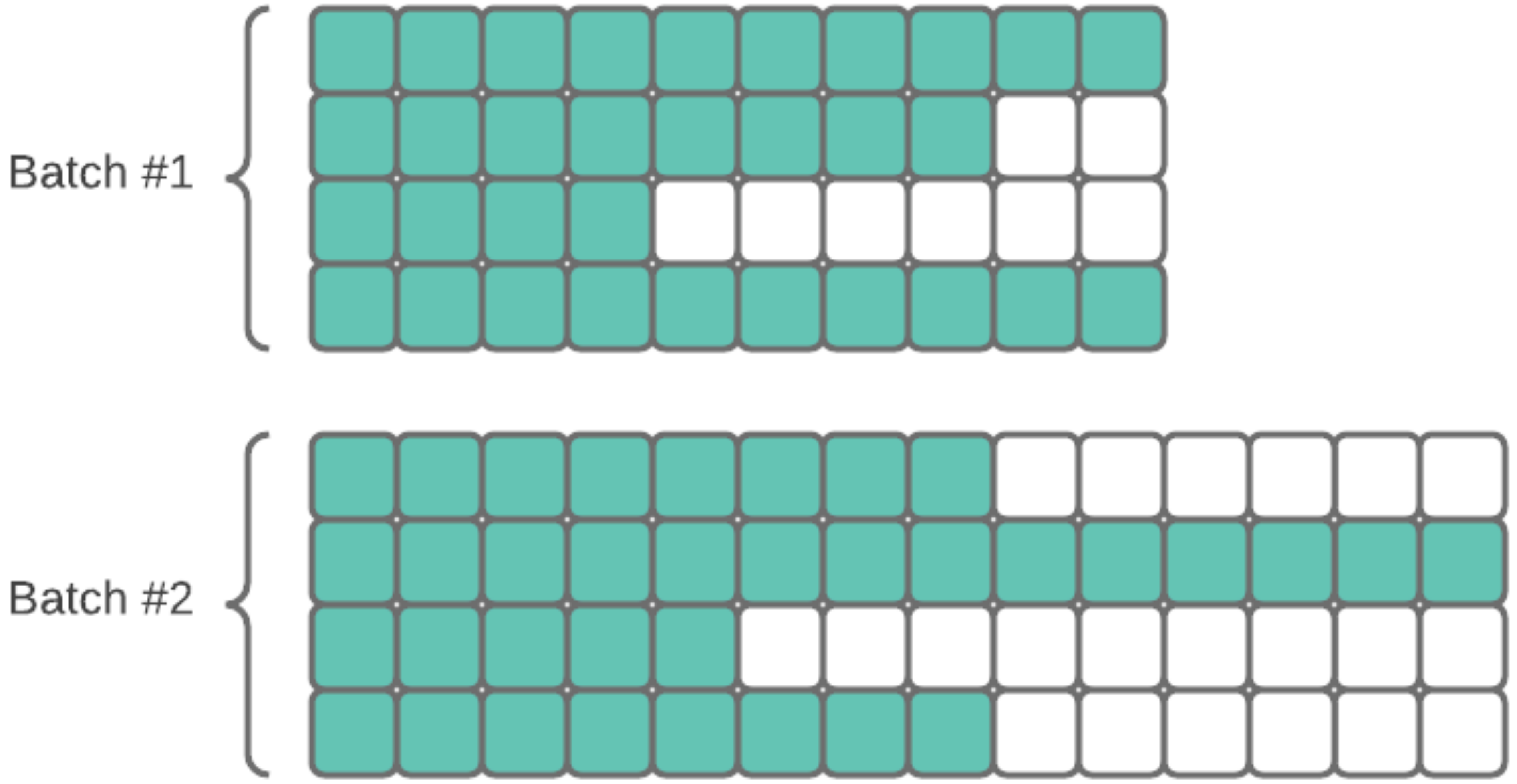
- When reading images, consider the code that reads them :)
  - Default PIL.Image.Open can be highly inefficient!
    Use at least Pillow-SIMD
  - Use better decoders (e.g. jpegturbo, nvJPEG from DALI)
- Heavy groups of augmentations can also slow you down
  - Consider moving them to GPU (e.g. kornia, DALI)
  - In most cases, you can switch to efficient implementations

Optimal sequence processing

- For sequential data, padding in batches is necessary
- However, padding the ENTIRE dataset can lead to redundant timesteps
- It’s usually better to store samples without padding and use collate_fn
- Also, bucket examples by length to further minimize padding
- …or, even pack multiple examples into the same sequence

Padded sequences sorted by decreasing lengths





Data pipelines: takeaways

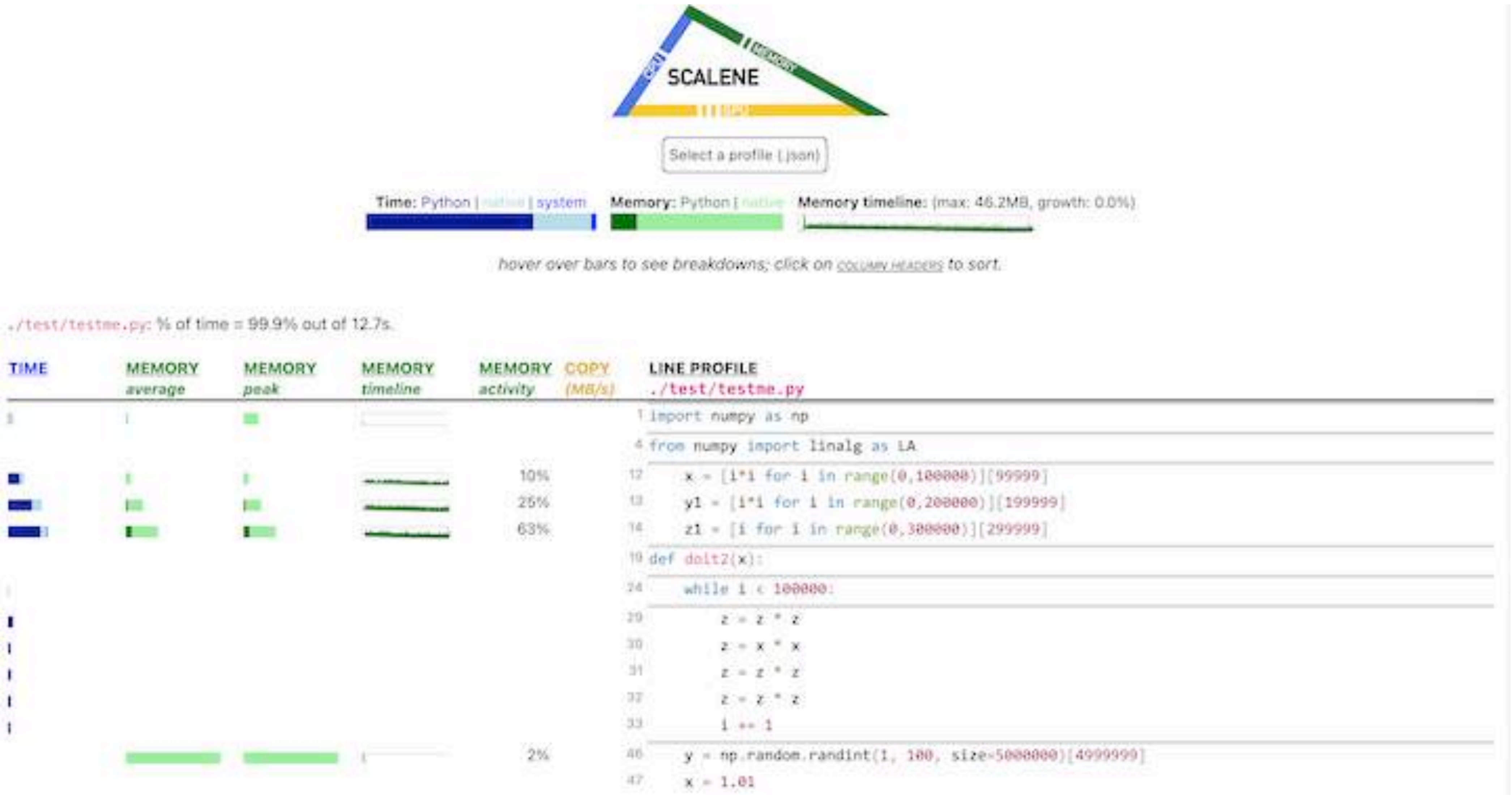
- Consider the performance/size of your storage when loading the data
- Use better deserialization primitives when available
- Try to avoid obvious inefficiencies when building task-specific pipelines

Profiling: what and why

- In benchmarking, we measure the speed of our program as a black box
- Profiling is a process of determining the runtime of parts of your program
- More of a “white box” approach

How to profile Python code?

- cProfile as a standard tool built into Python
- Sampling-based profilers (scalene etc.)
- Some of them (e.g. py-spy) even allow to attach to running code!

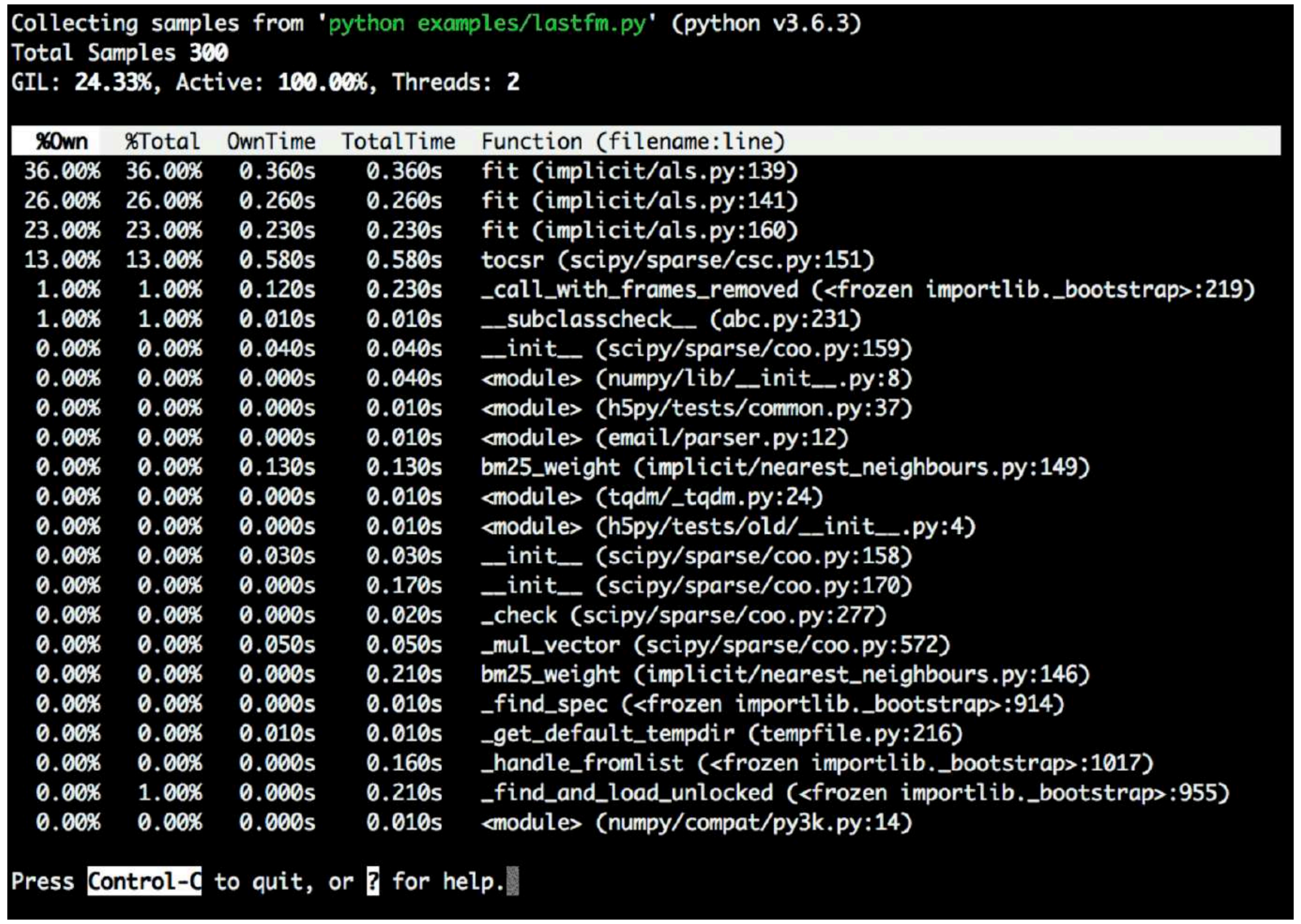


Time: Python | native | system Memory: Python | native Memory timeline: (max: 46.2MB, growth: 0.0%)

hover over bars to see breakdowns; click on COLUMN HEADERS *to sort.*

./test/testme.py: % of time = 99.9% out of 12.7s.

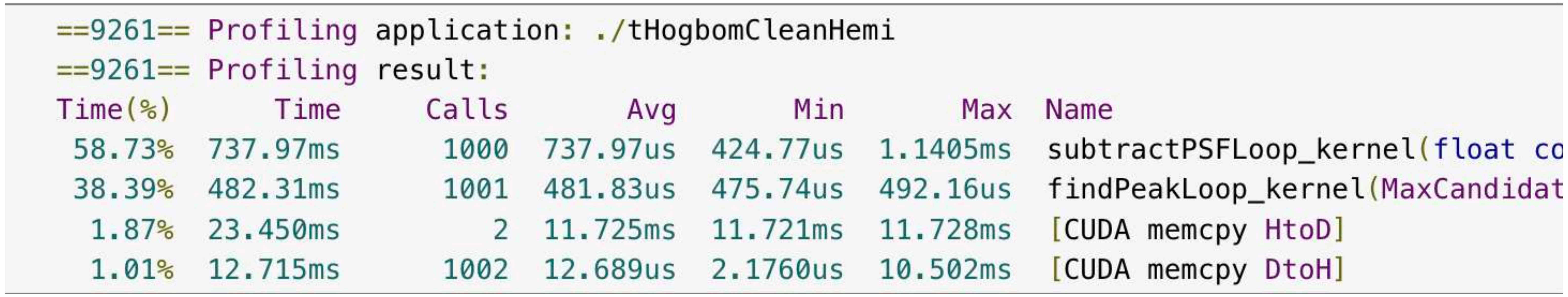
```
TIME   MEMORY    MEMORY   MEMORY     MEMORY    COPY     LINE PROFILE
       average   peak     timeline   activity  (MB/s)   ./test/testme.py
                                                         1 import numpy as np
                                                         4 from numpy import linalg as LA
                                     10%                12     x = [i*i for i in range(0,100000)][99999]
                                     25%                13     y1 = [i*i for i in range(0,200000)][199999]
                                     63%                14     z1 = [i for i in range(0,300000)][299999]
                                                        19 def doit2(x):
                                                        24     while i < 100000:
                                                        29         z = z * z
                                                        30         z = x * x
                                                        31         z = z * z
                                                        32         z = z * z
                                                        33         i += 1
                                     2%                 46     y = np.random.randint(1, 100, size=5000000)[4999999]
                                                        47     x = 1.01
```



```
Collecting samples from 'python examples/lastfm.py' (python v3.6.3)
Total Samples 300
GIL: 24.33%, Active: 100.00%, Threads: 2

  %Own   %Total  OwnTime  TotalTime  Function (filename:line)
 36.00%  36.00%   0.360s    0.360s   fit (implicit/als.py:139)
 26.00%  26.00%   0.260s    0.260s   fit (implicit/als.py:141)
 23.00%  23.00%   0.230s    0.230s   fit (implicit/als.py:160)
 13.00%  13.00%   0.580s    0.580s   tocsr (scipy/sparse/csc.py:151)
  1.00%   1.00%   0.120s    0.230s   _call_with_frames_removed (<frozen importlib._bootstrap>:219)
  1.00%   1.00%   0.010s    0.010s   __subclasscheck__ (abc.py:231)
  0.00%   0.00%   0.040s    0.040s   __init__ (scipy/sparse/coo.py:159)
  0.00%   0.00%   0.000s    0.040s   <module> (numpy/lib/__init__.py:8)
  0.00%   0.00%   0.000s    0.010s   <module> (h5py/tests/common.py:37)
  0.00%   0.00%   0.000s    0.010s   <module> (email/parser.py:12)
  0.00%   0.00%   0.130s    0.130s   bm25_weight (implicit/nearest_neighbours.py:149)
  0.00%   0.00%   0.000s    0.010s   <module> (tqdm/_tqdm.py:24)
  0.00%   0.00%   0.000s    0.010s   <module> (h5py/tests/old/__init__.py:4)
  0.00%   0.00%   0.030s    0.030s   __init__ (scipy/sparse/coo.py:158)
  0.00%   0.00%   0.000s    0.170s   __init__ (scipy/sparse/coo.py:170)
  0.00%   0.00%   0.000s    0.020s   _check (scipy/sparse/coo.py:277)
  0.00%   0.00%   0.050s    0.050s   _mul_vector (scipy/sparse/coo.py:572)
  0.00%   0.00%   0.000s    0.210s   bm25_weight (implicit/nearest_neighbours.py:146)
  0.00%   0.00%   0.000s    0.010s   _find_spec (<frozen importlib._bootstrap>:914)
  0.00%   0.00%   0.010s    0.010s   _get_default_tempdir (tempfile.py:216)
  0.00%   0.00%   0.000s    0.160s   _handle_fromlist (<frozen importlib._bootstrap>:1017)
  0.00%   1.00%   0.000s    0.210s   _find_and_load_unlocked (<frozen importlib._bootstrap>:955)
  0.00%   0.00%   0.000s    0.010s   <module> (numpy/compat/py3k.py:14)

Press Control-C to quit, or ? for help.
```

How to profile GPU code?

- nvprof is the low-level profiling tool
- Gives you the performance of low-level kernel launches and copies



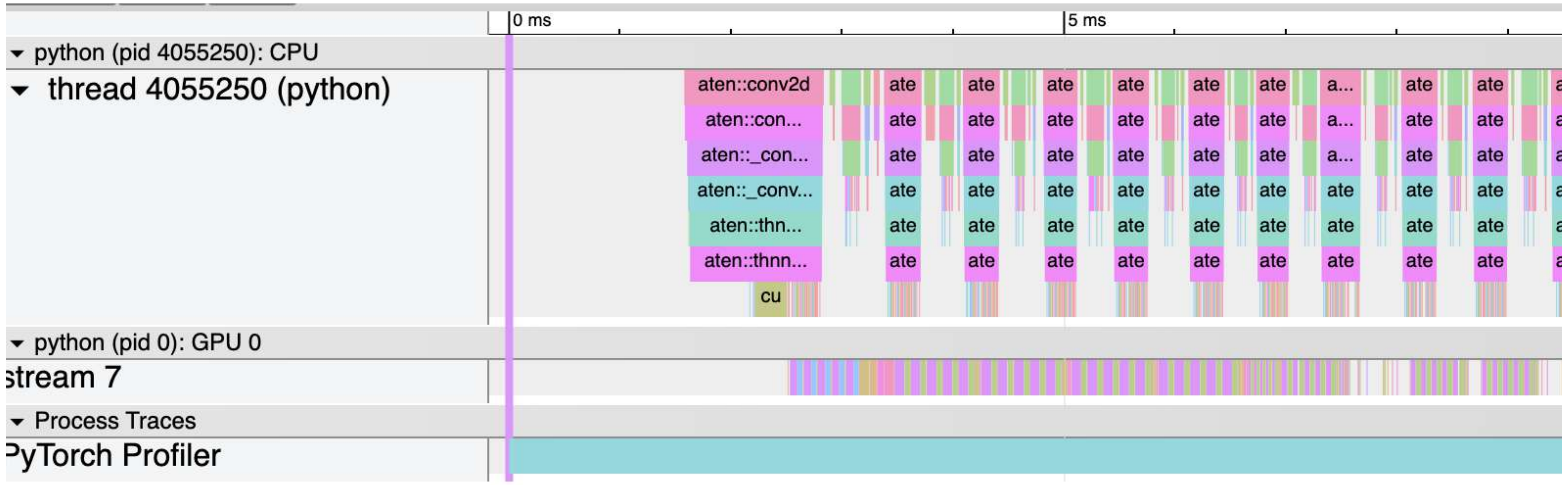
```
==9261== Profiling application: ./tHogbomCleanHemi
==9261== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
 58.73%  737.97ms      1000  737.97us  424.77us  1.1405ms  subtractPSFLoop_kernel(float co
 38.39%  482.31ms      1001  481.83us  475.74us  492.16us  findPeakLoop_kernel(MaxCandidat
  1.87%  23.450ms         2  11.725ms  11.721ms  11.728ms  [CUDA memcpy HtoD]
  1.01%  12.715ms      1002  12.689us  2.1760us  10.502ms  [CUDA memcpy DtoH]
```

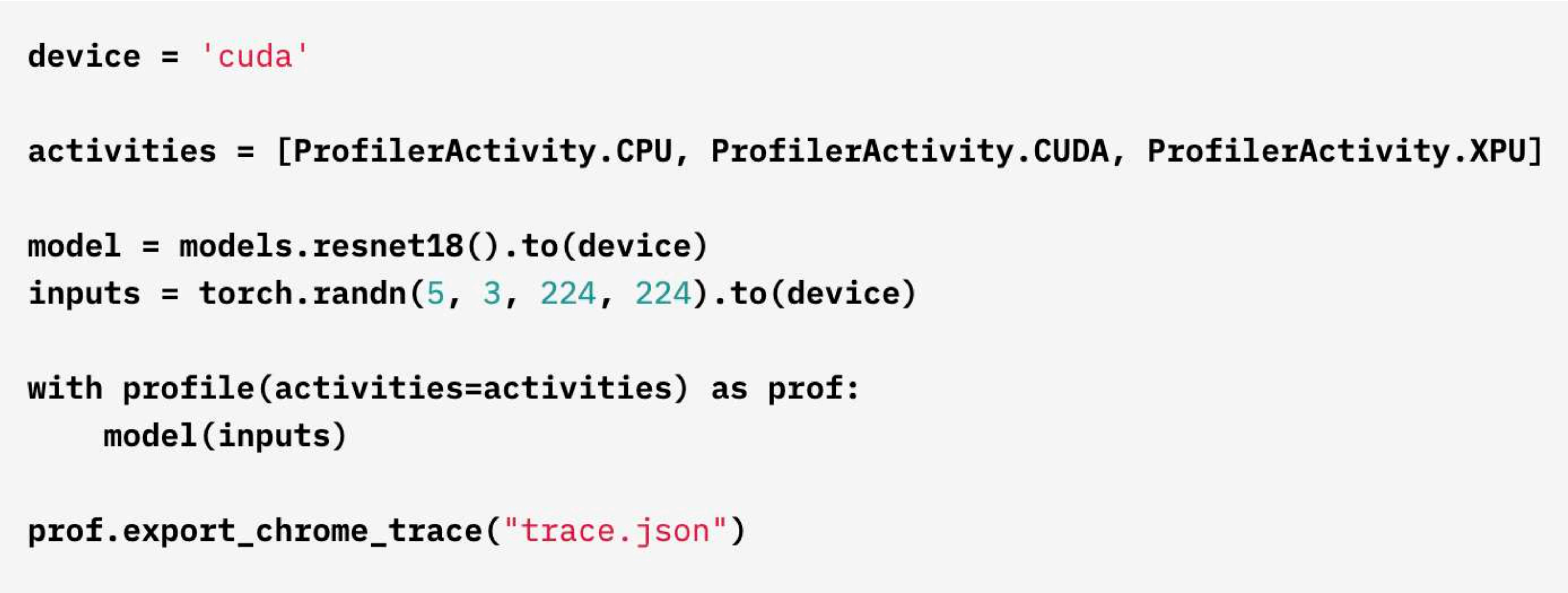
developer.nvidia.com/blog/cuda-pro-tip-nvprof-your-handy-universal-gpu-profiler/

How to profile PyTorch code?

- High-level: torch.utils.bottleneck
- Older API: torch.autograd.profiler
- Newer one: torch.profiler

PyTorch Profiler + trace viewer

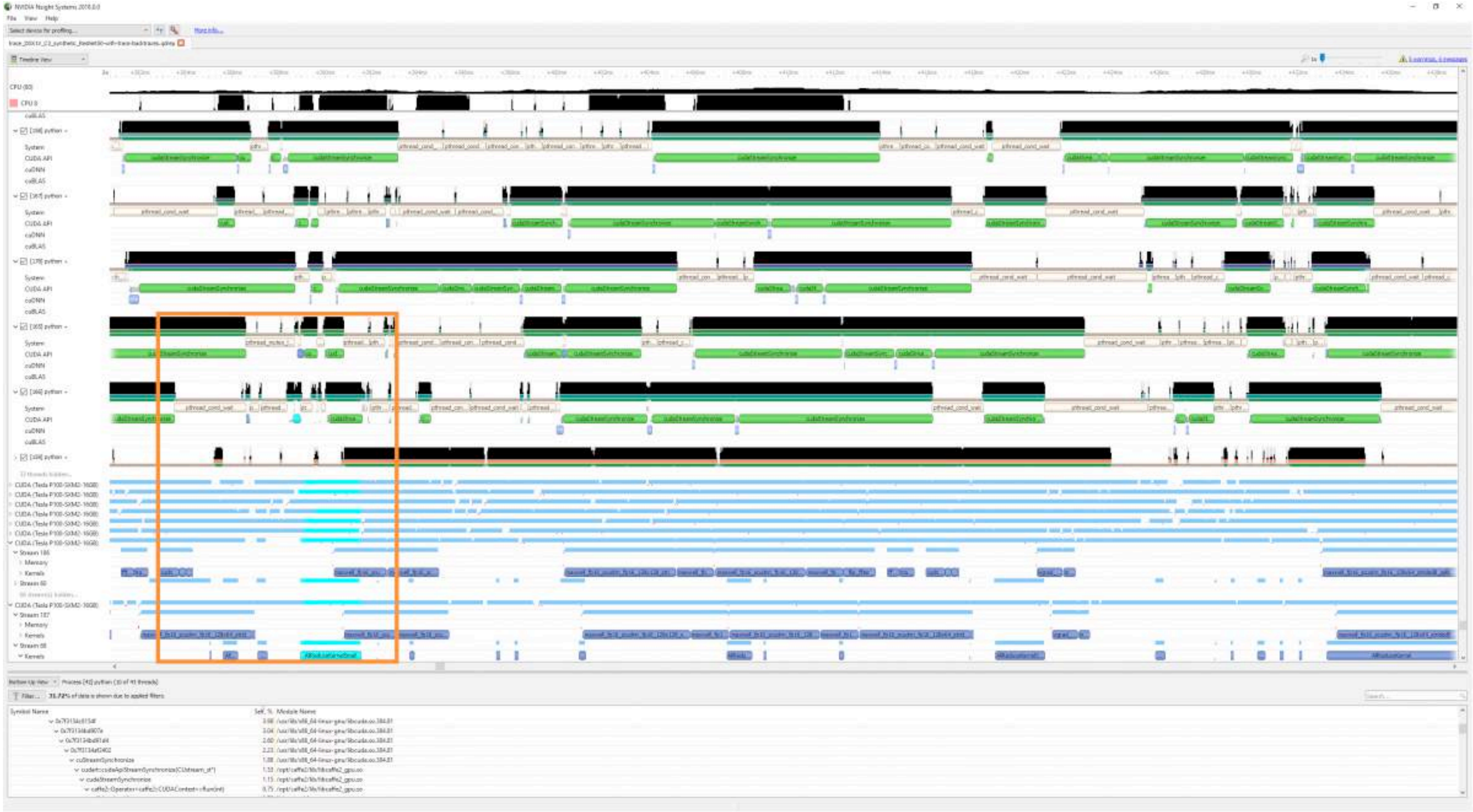




```
device = 'cuda'

activities = [ProfilerActivity.CPU, ProfilerActivity.CUDA, ProfilerActivity.XPU]

model = models.resnet18().to(device)
inputs = torch.randn(5, 3, 224, 224).to(device)

with profile(activities=activities) as prof:
    model(inputs)

prof.export_chrome_trace("trace.json")
```

pytorch.org/tutorials/recipes/recipes/profiler_recipe.html

Nsight Systems/Nsight Compute



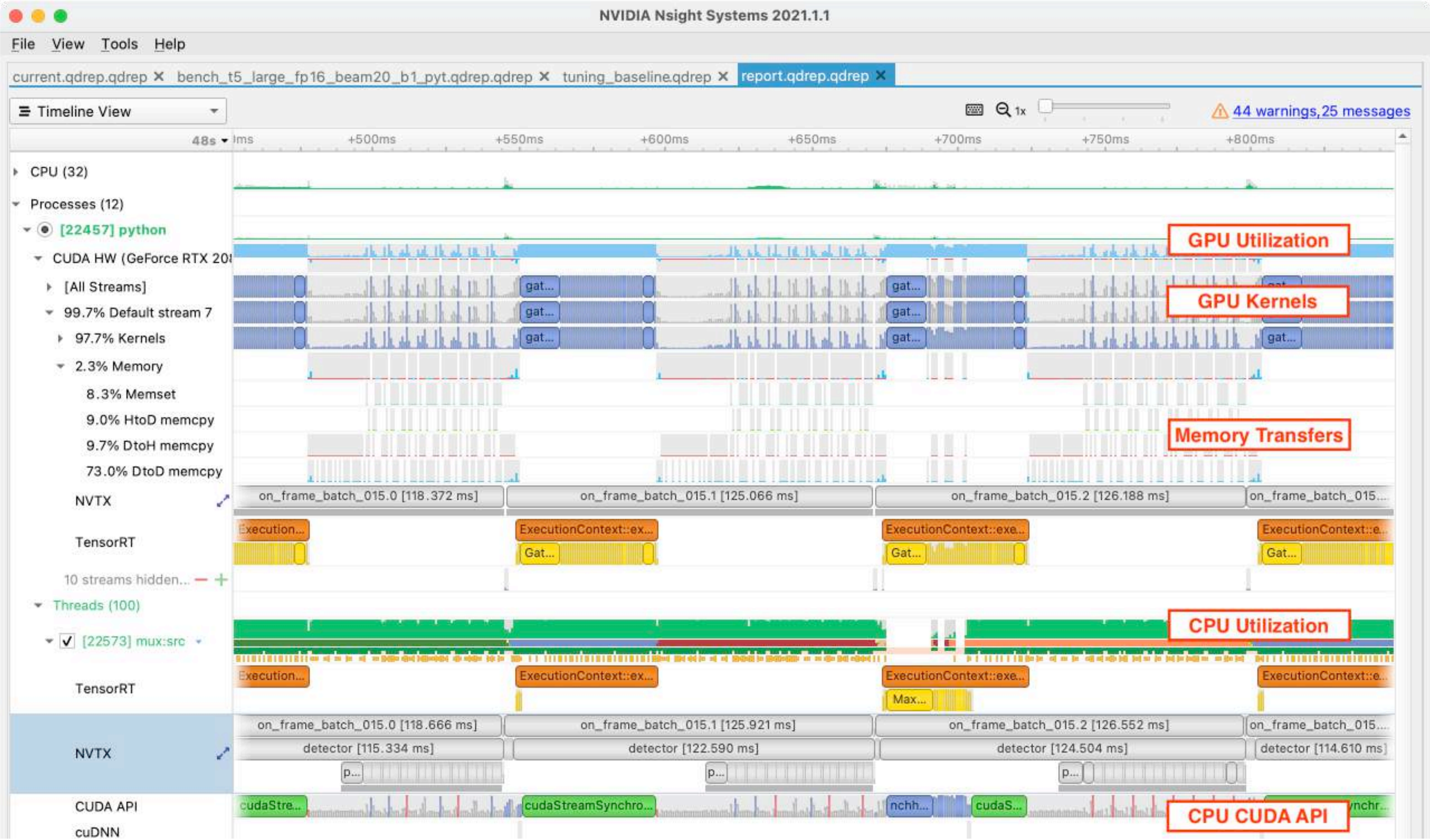
Process [42] python (10 of 41 threads)

31.72% of data is shown due to applied filters

```
Symbol Name                                          Self, %  Module Name
 v 0x7f3134c0154f                                     3.98    /usr/lib/x86_64-linux-gnu/libcuda.so.384.81
  v 0x7f3134bd907e                                    3.04    /usr/lib/x86_64-linux-gnu/libcuda.so.384.81
   v 0x7f3134bd91d4                                   2.60    /usr/lib/x86_64-linux-gnu/libcuda.so.384.81
    v 0x7f3134af2402                                  2.23    /usr/lib/x86_64-linux-gnu/libcuda.so.384.81
     v cuStreamSynchronize                            1.88    /usr/lib/x86_64-linux-gnu/libcuda.so.384.81
      v cudart::cudaApiStreamSynchronize(CUstream_st*) 1.33   /opt/caffe2/lib/libcaffe2_gpu.so
       v cudaStreamSynchronize                        1.15    /opt/caffe2/lib/libcaffe2_gpu.so
        v caffe2::Operator<caffe2::CUDAContext>::Run(int) 0.75 /opt/caffe2/lib/libcaffe2_gpu.so
```

developer.download.nvidia.com/video/gputechconf/gtc/2019/presentation/s9339-profiling-deep-learning-networks.pdf

Profiling: typical patterns



GPU Utilization

GPU Kernels

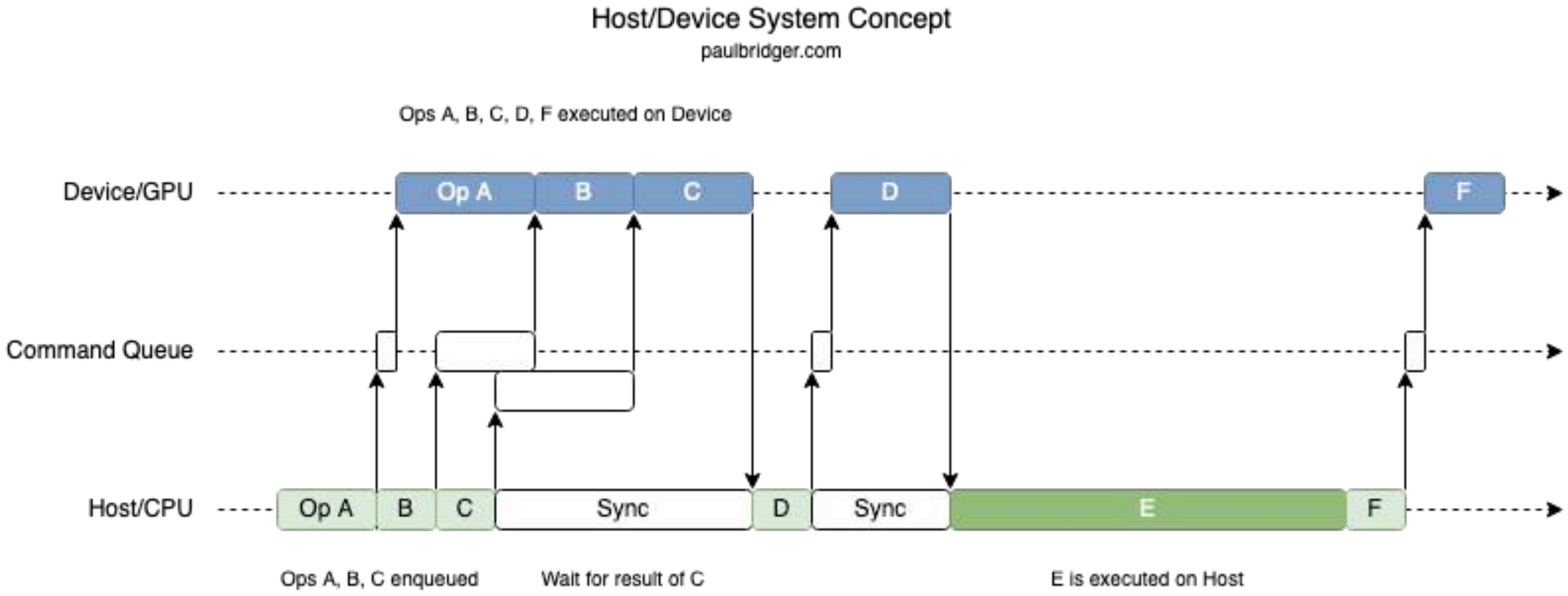
Memory Transfers

CPU Utilization

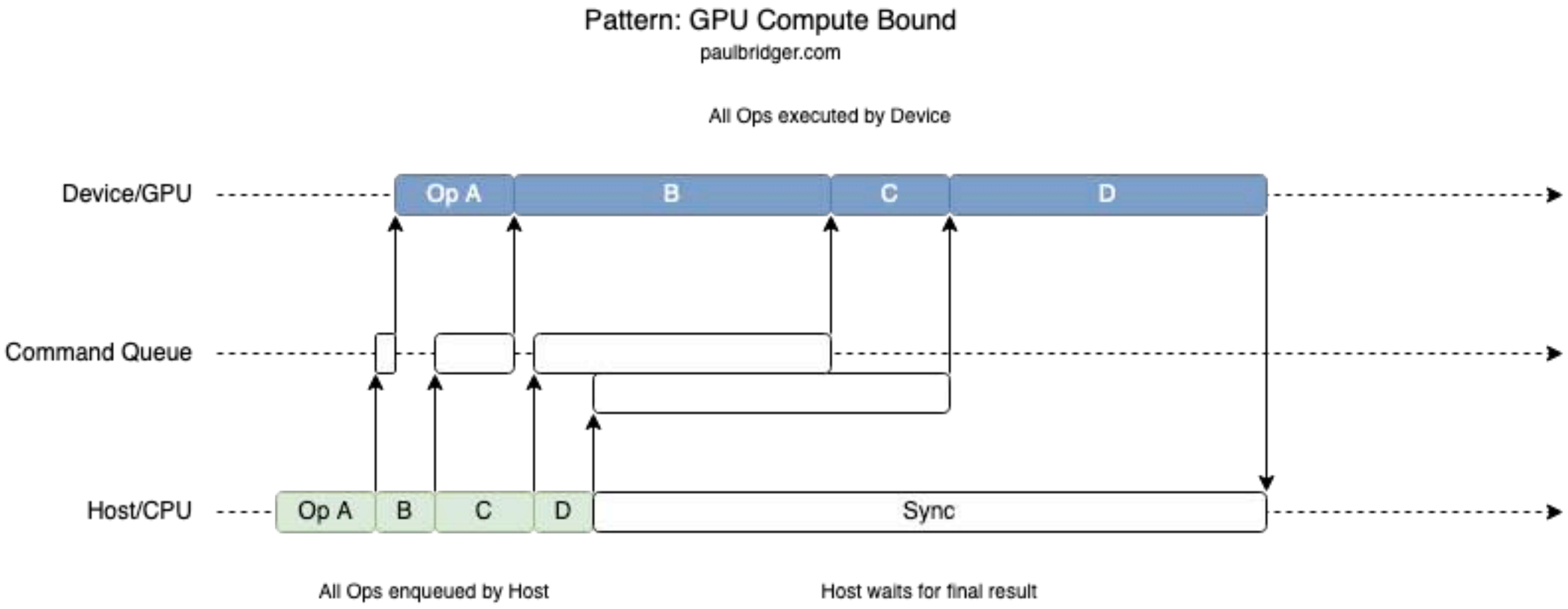
CPU CUDA API

paulbridger.com/posts/nsight-systems-systematic-optimization

Profiling: typical patterns

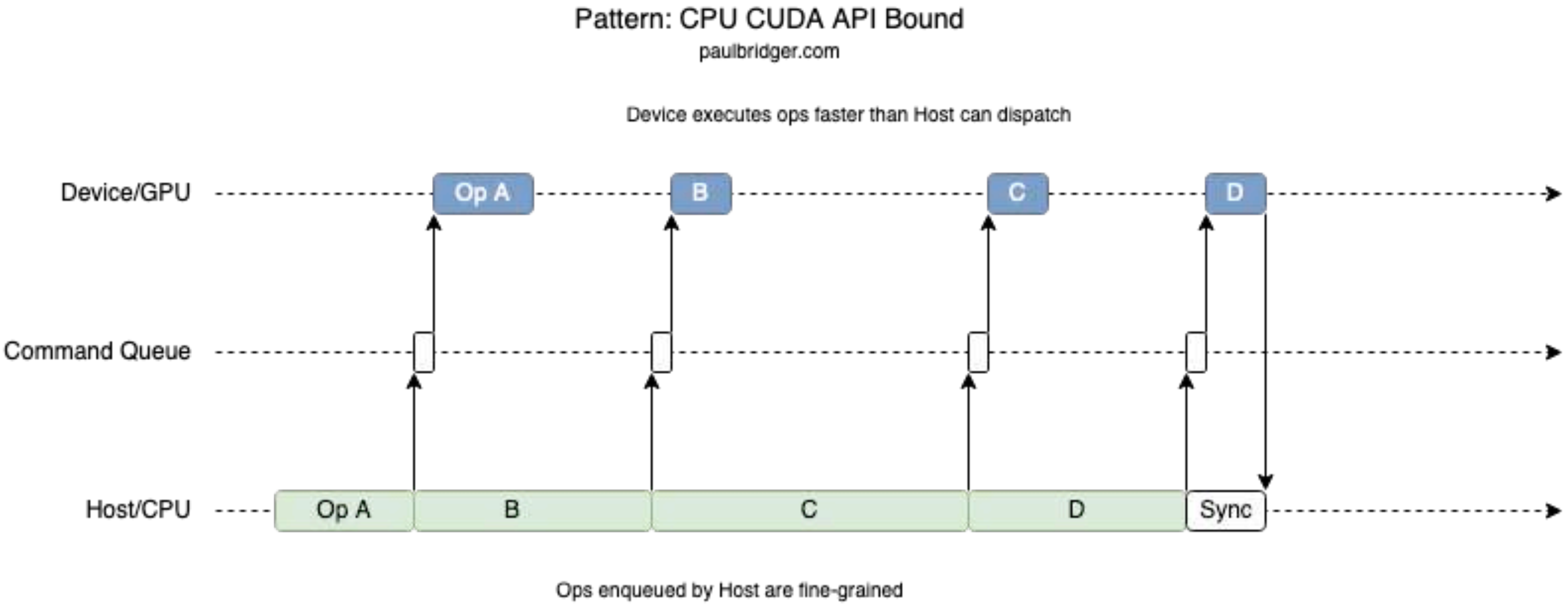


paulbridger.com/posts/nsight-systems-systematic-optimization



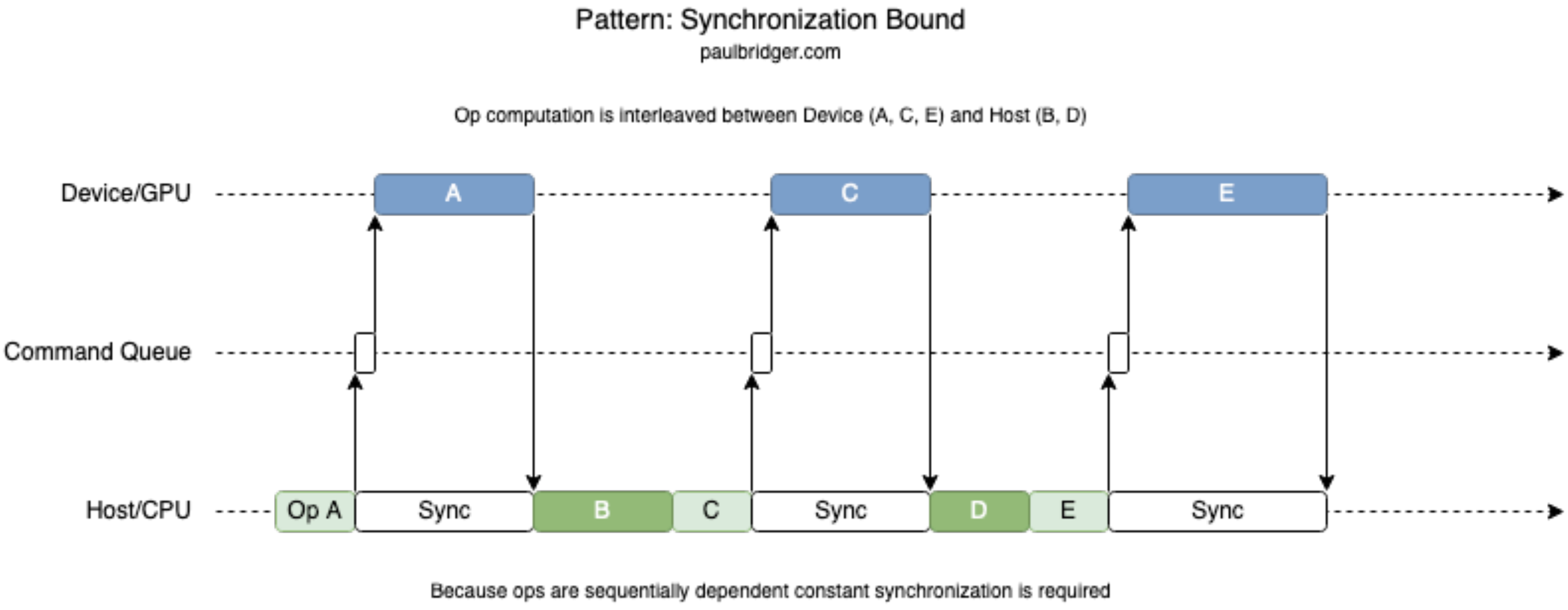
- This is the best case!
- You need to optimize the model itself (lower precision, faster kernels etc)

paulbridger.com/posts/nsight-systems-systematic-optimization


Pattern: CPU CUDA API Bound
paulbridger.com
Device executes ops faster than Host can dispatch
Device/GPU
Op A
B
C
D
Command Queue
Host/CPU
Op A
B
C
D
Sync
Ops enqueued by Host are fine-grained

- Operations run faster than kernels are scheduled
- Also happens during inference
- You need to optimize the CUDA API calls (torch.compile, TorchScript) or have more compute-intensive operations (e.g. larger batches)

paulbridger.com/posts/nsight-systems-systematic-optimization



Pattern: Synchronization Bound

paulbridger.com

Op computation is interleaved between Device (A, C, E) and Host (B, D)

Because ops are sequentially dependent constant synchronization is required

- CPU and GPU processing are too heavily interleaved
- Remove unnecessary synchronization points, execute as much work on the GPU as possible

paulbridger.com/posts/nsight-systems-systematic-optimization

Profiling: takeaways

- A very useful tool for understanding the performance of your pipeline
- Can be applied to both CPU and GPU code
- Depending on the required granularity of measurements,
  you can use different approaches