Efficient Deep Learning Systems
Optimizing training pipelines

Max Ryabinin

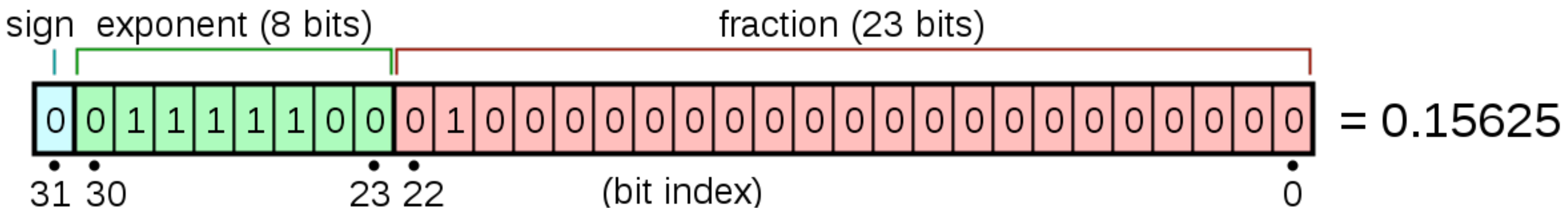
2022

Plan for today

- Mixed precision training
    - When and why to use it
    - How to enable it and utilize it to the fullest
    - Dealing with stability in training
- Training pipeline optimization
    - Hardware considerations
    - Storing and loading data efficiently
- Profiling DL code

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



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

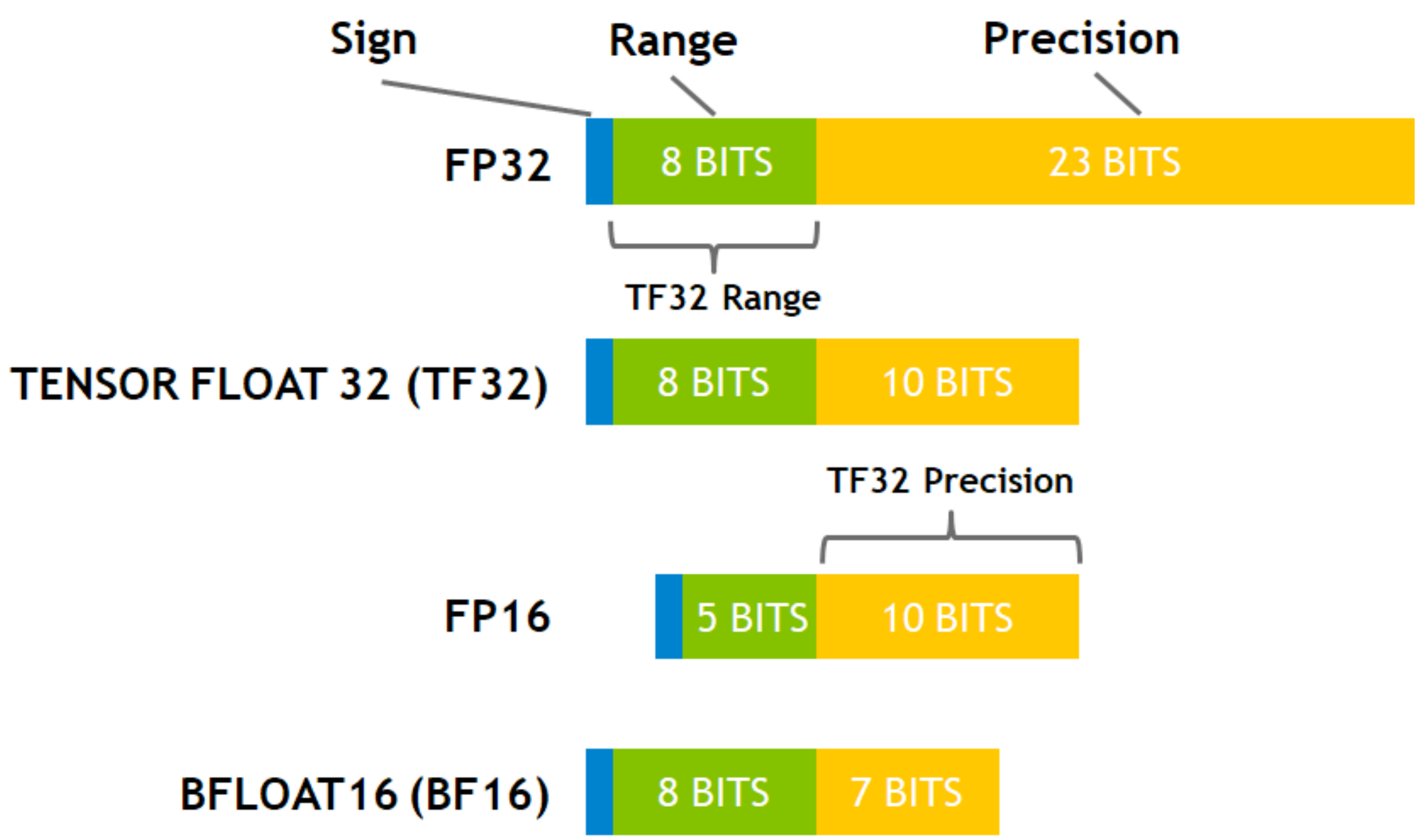
- Special values (0, NaN, ±inf) are encoded by exponent values

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

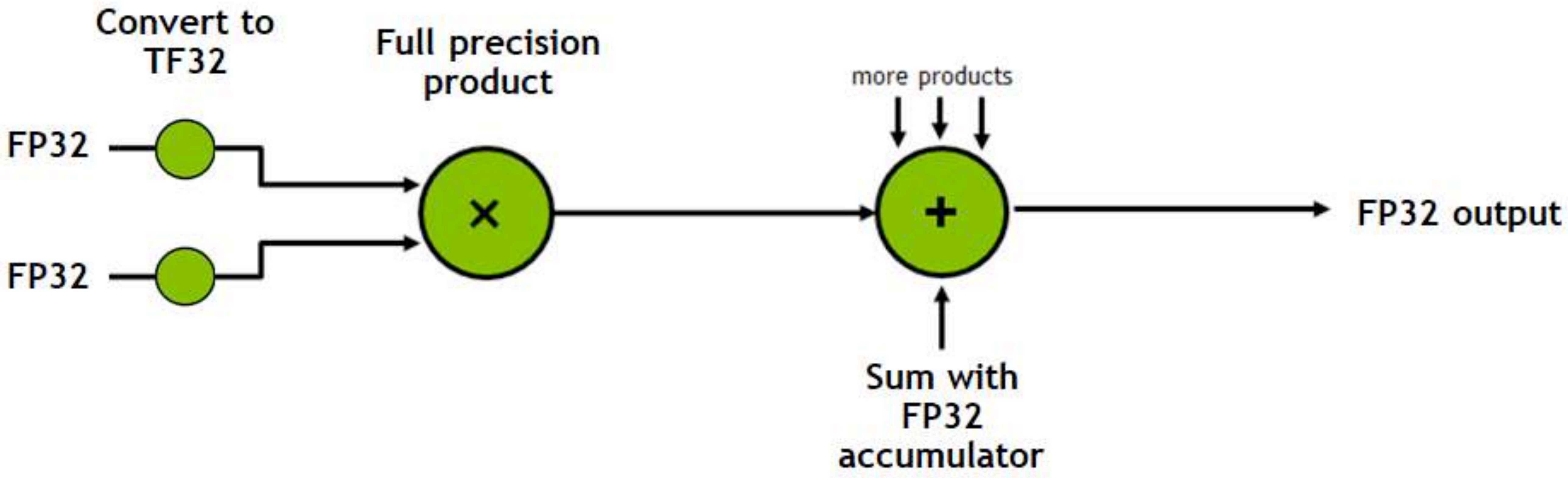
- Naive FP16 is not the only option!
- Specialized formats preserve dynamic range for computations



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

Switching to lower precision

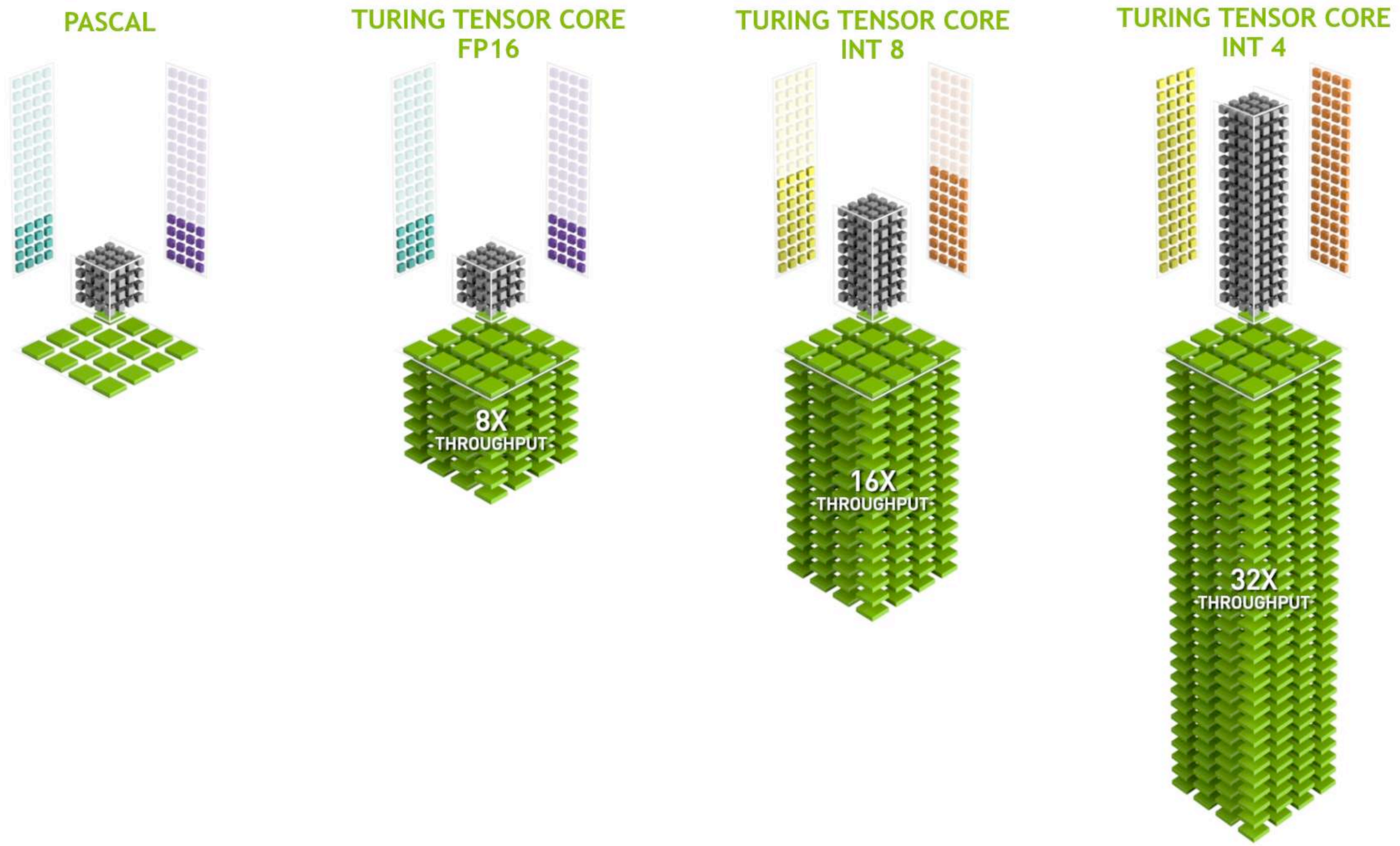
- FP16 exists since CUDA 8, just allocate the tensor/cast it to `half`
- BF16 is available on CPUs and TPUs [1], `Tensor.bfloat16()` in PyTorch
- TF32 is enabled for you on the Ampere GPUs
  - Never exposed as a data type, only as a type for specific operations [2]



[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)



https://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!

https://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:

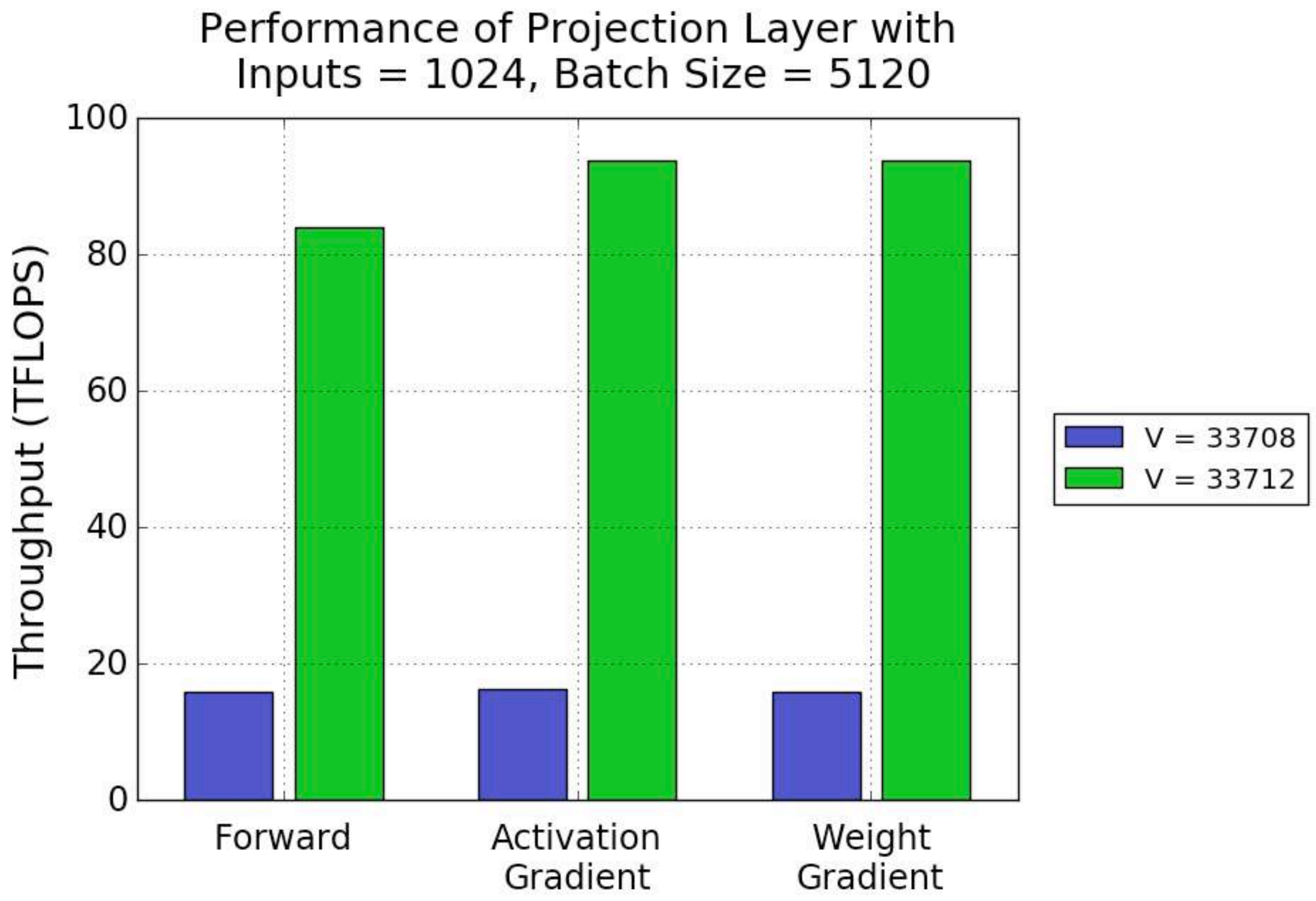
Table 1. Tensor Core requirements by cuBLAS or cuDNN version for some common data precisions. These requirements apply to matrix dimensions M, N, and K.

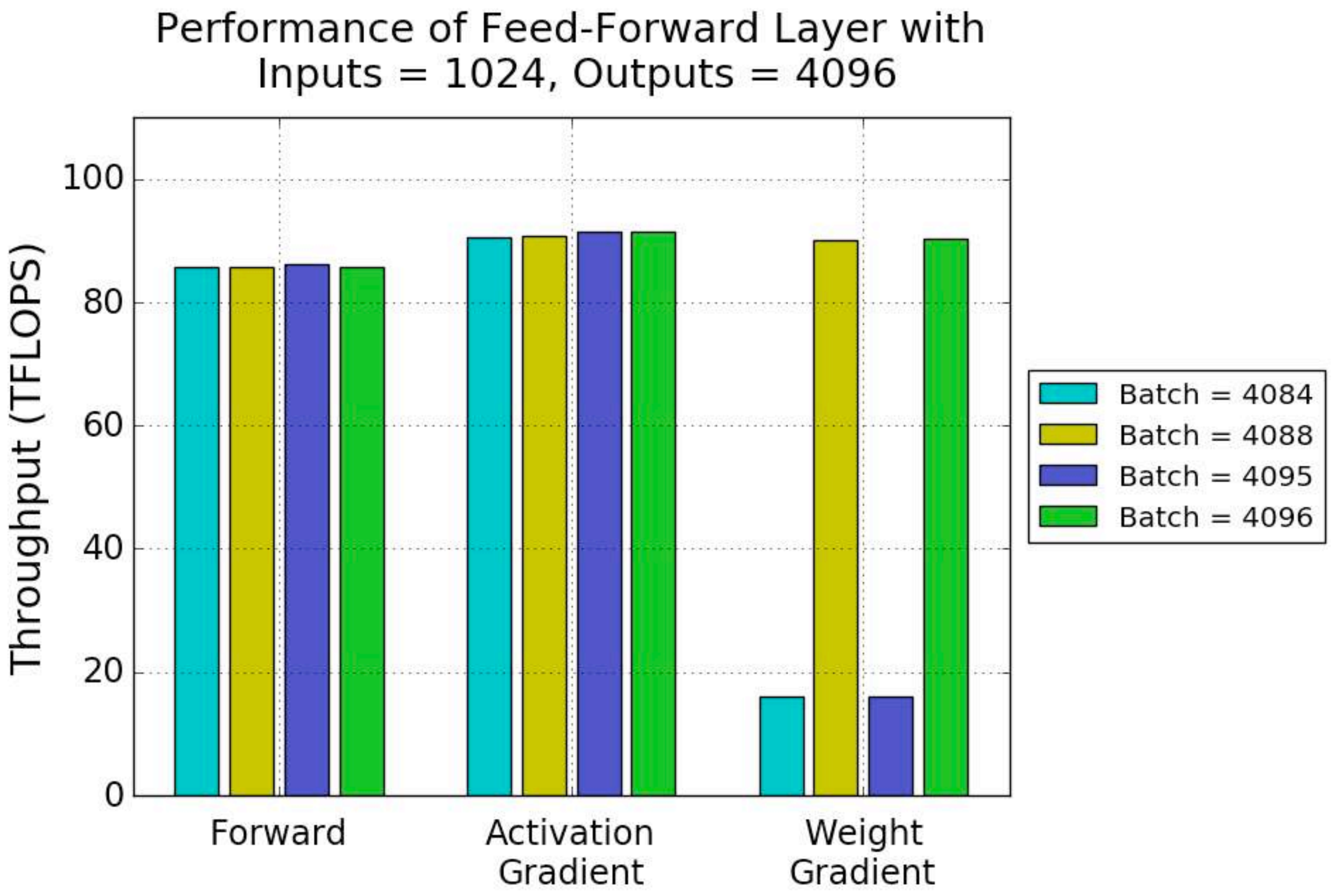
Tensor Cores can be used for...	cuBLAS version < 11.0 cuDNN version < 7.6.3	cuBLAS version ≥ 11.0 cuDNN version ≥ 7.6.3
INT8	Multiples of 16	Always but most efficient with multiples of 16; on A100, multiples of 128.
FP16	Multiples of 8	Always but most efficient with multiples of 8; on A100, multiples 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] https://docs.nvidia.com/deeplearning/performance/dl-performance-matrix-multiplication/index.html#requirements-tc
[2] https://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] https://docs.nvidia.com/deeplearning/performance/dl-performance-matrix-multiplication/index.html#requirements-tc
[2] https://developer.download.nvidia.com/video/gputechconf/gtc/2019/presentation/s9926-tensor-core-performance-the-ultimate-guide.pdf

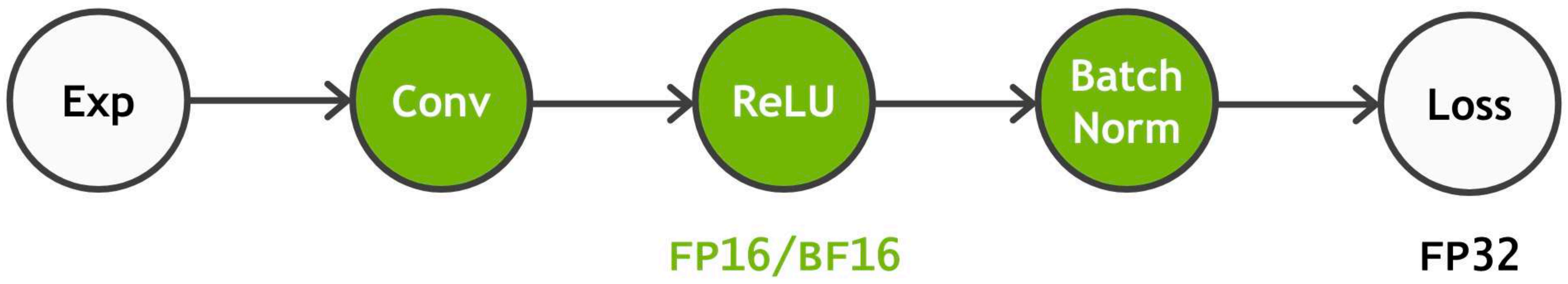
Utilizing Tensor Cores

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

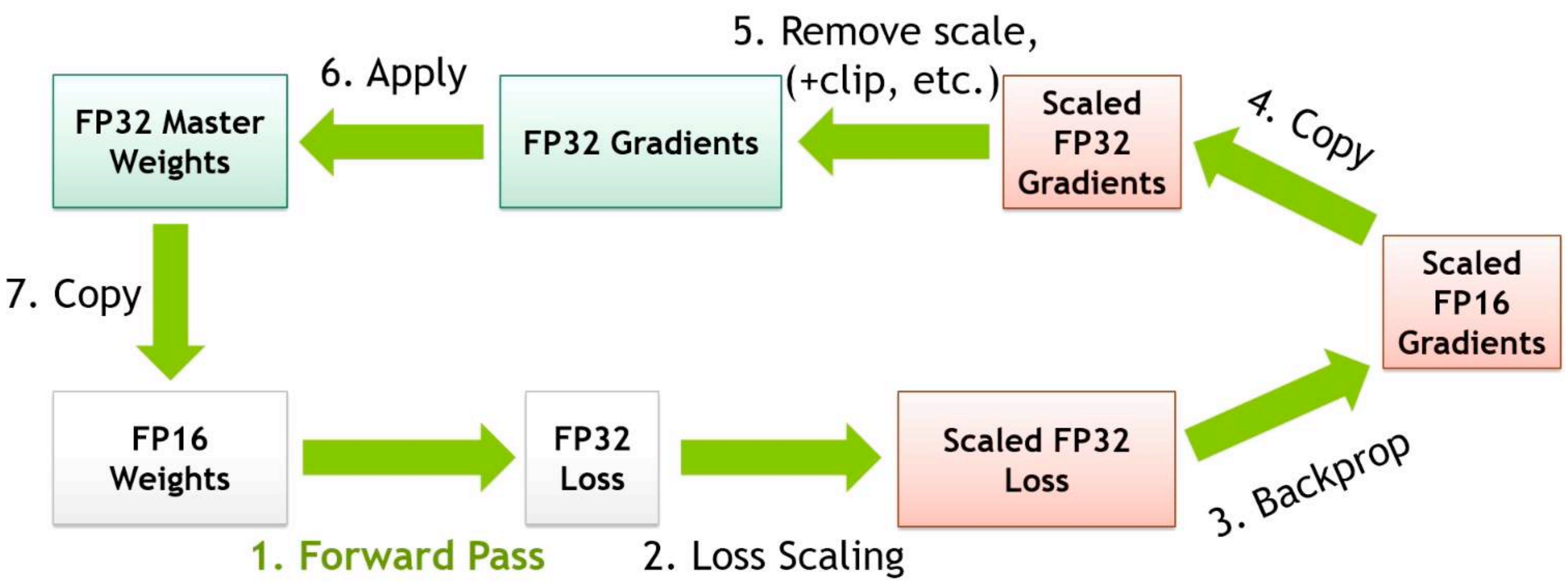
Mixed precision training

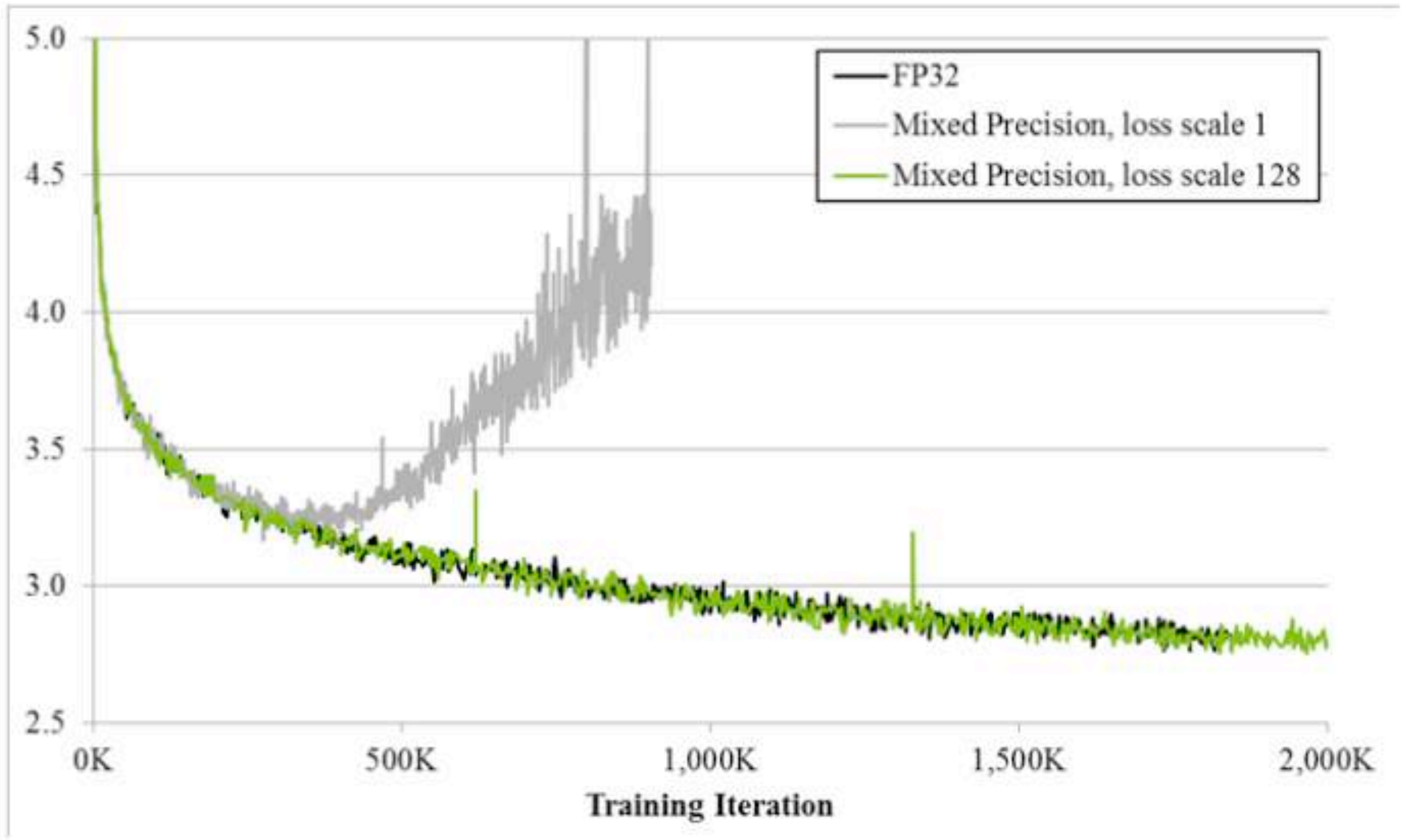
- Training in pure FP16 hardly works
- Some operations (e.g. matrix multiplication) can work, others (softmax, batch normalization) need higher precision
- Mixed precision training casts layer activations to appropriate data types
- Supported in popular DL frameworks (e.g. torch.cuda.amp)



Loss scaling

- To prevent underflows, we need to scale the FP16 gradients by a small number
- Ironically, this can lead to overflows when unscaling
- Dynamic loss scaling detects such overflows and repeatedly halves the scale





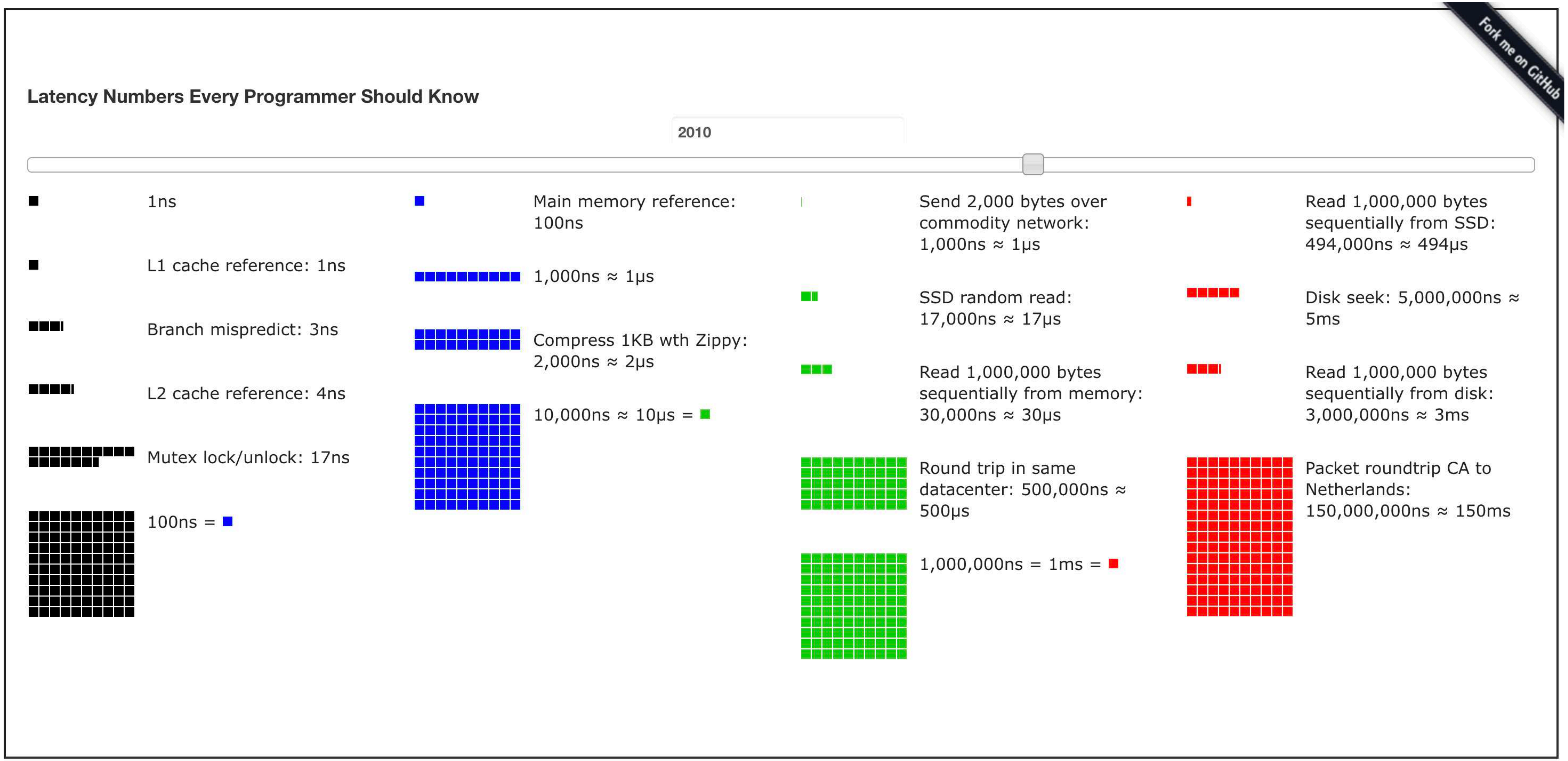
AMP: takeaways

- Use more efficient data types when available
- Mind the sizes/operation types
- In most cases, this is enabled for you

Bottlenecks in data loading

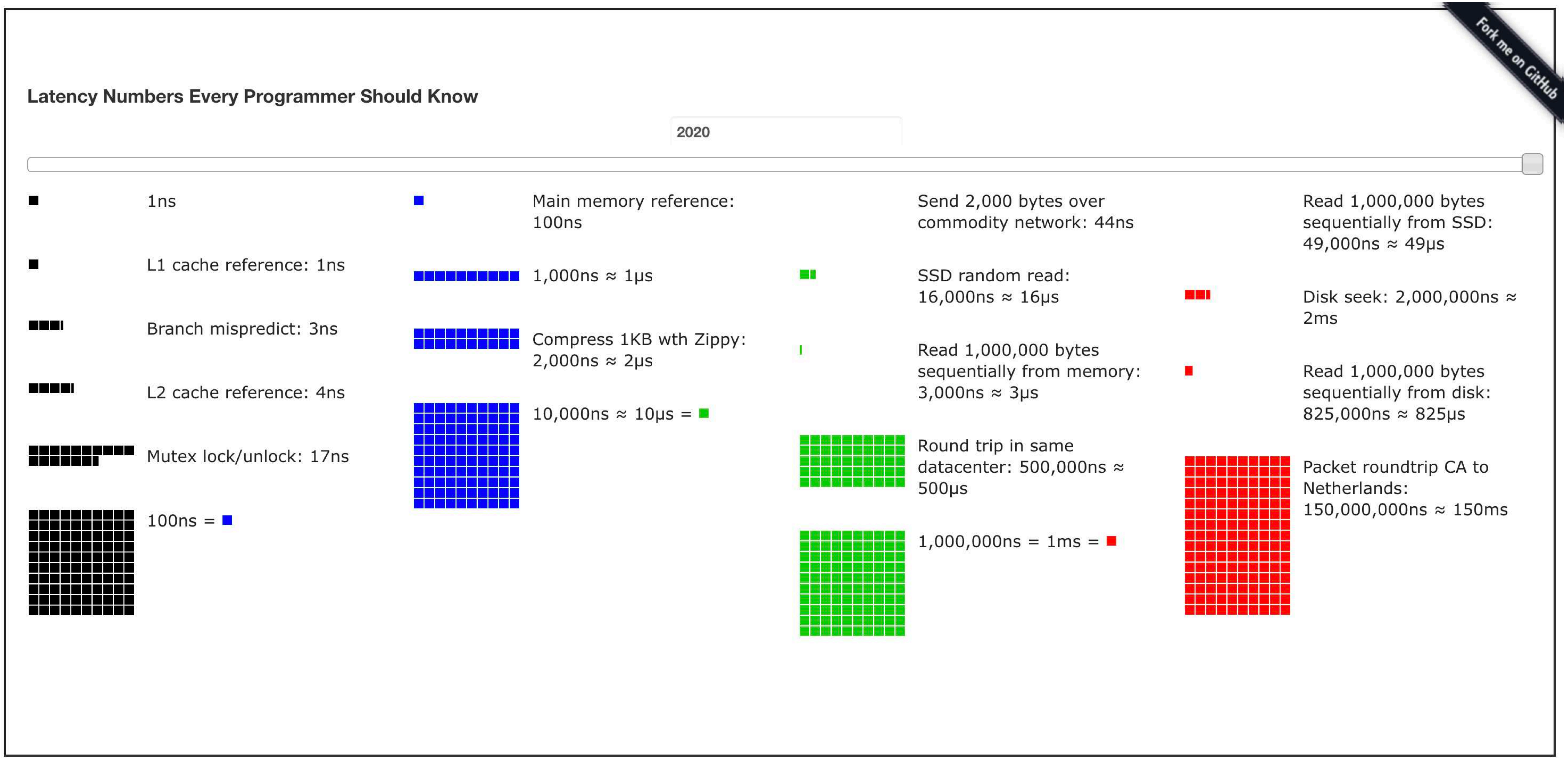
https://colin-scott.github.io/personal_website/research/interactive_latency.html

Bottlenecks in data loading



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

Bottlenecks in data loading



https://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

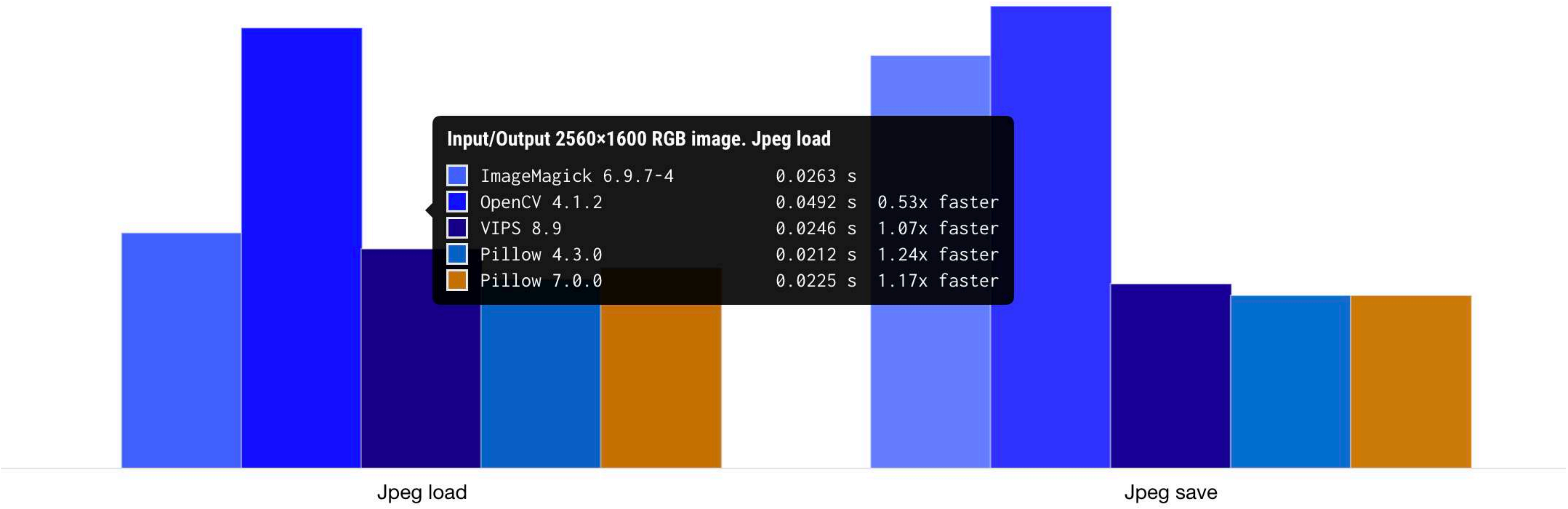
- Raw files are often easy to visualize, but storage-inefficient (especially when accessing external storage)
- In some cases, you might benefit from better formats:
    - For structured data, Apache Arrow/Protobuf/msgpack etc.
    - For images, apply non-random “heavy” processing before training
  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

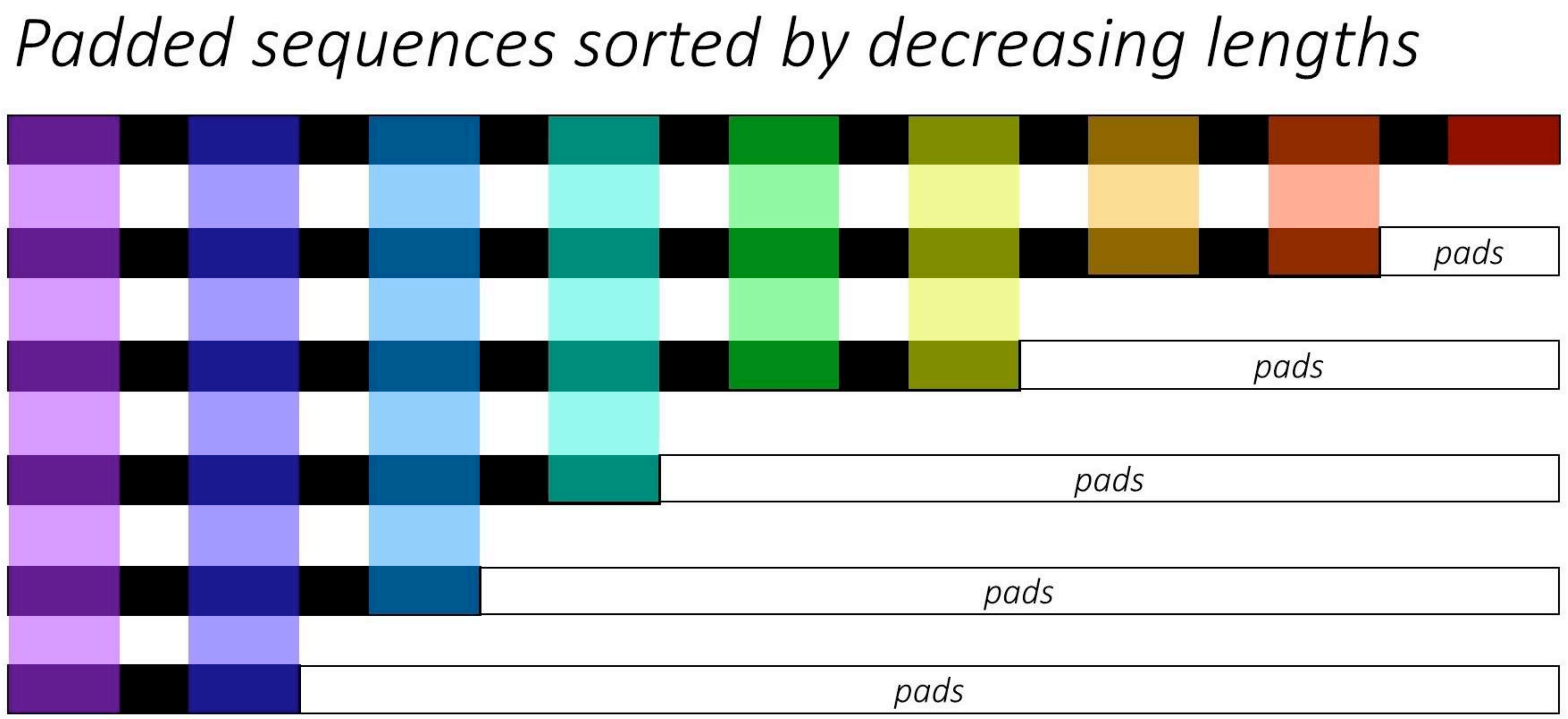


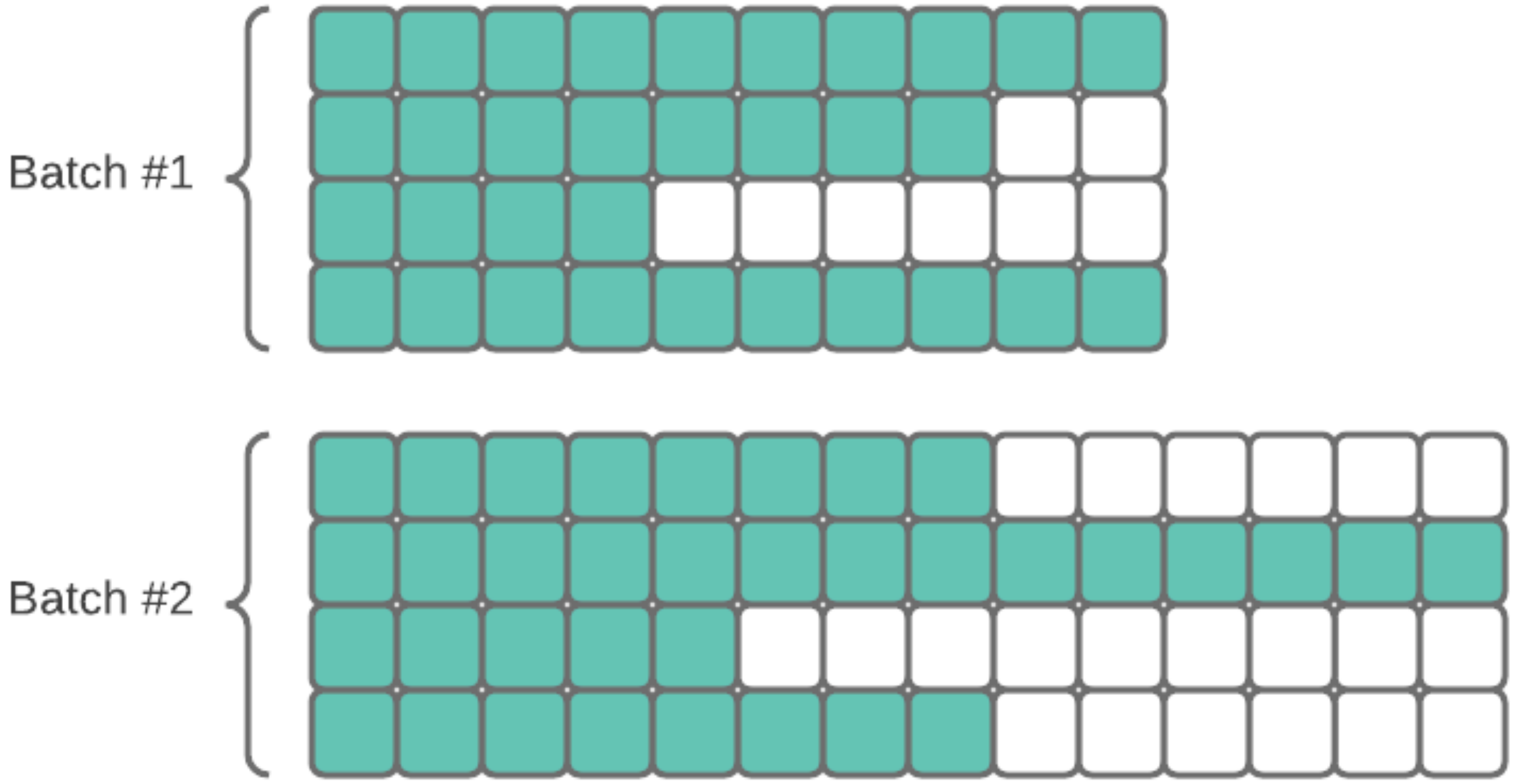
Performance of image loading

- 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





Data pipelines: takeaways

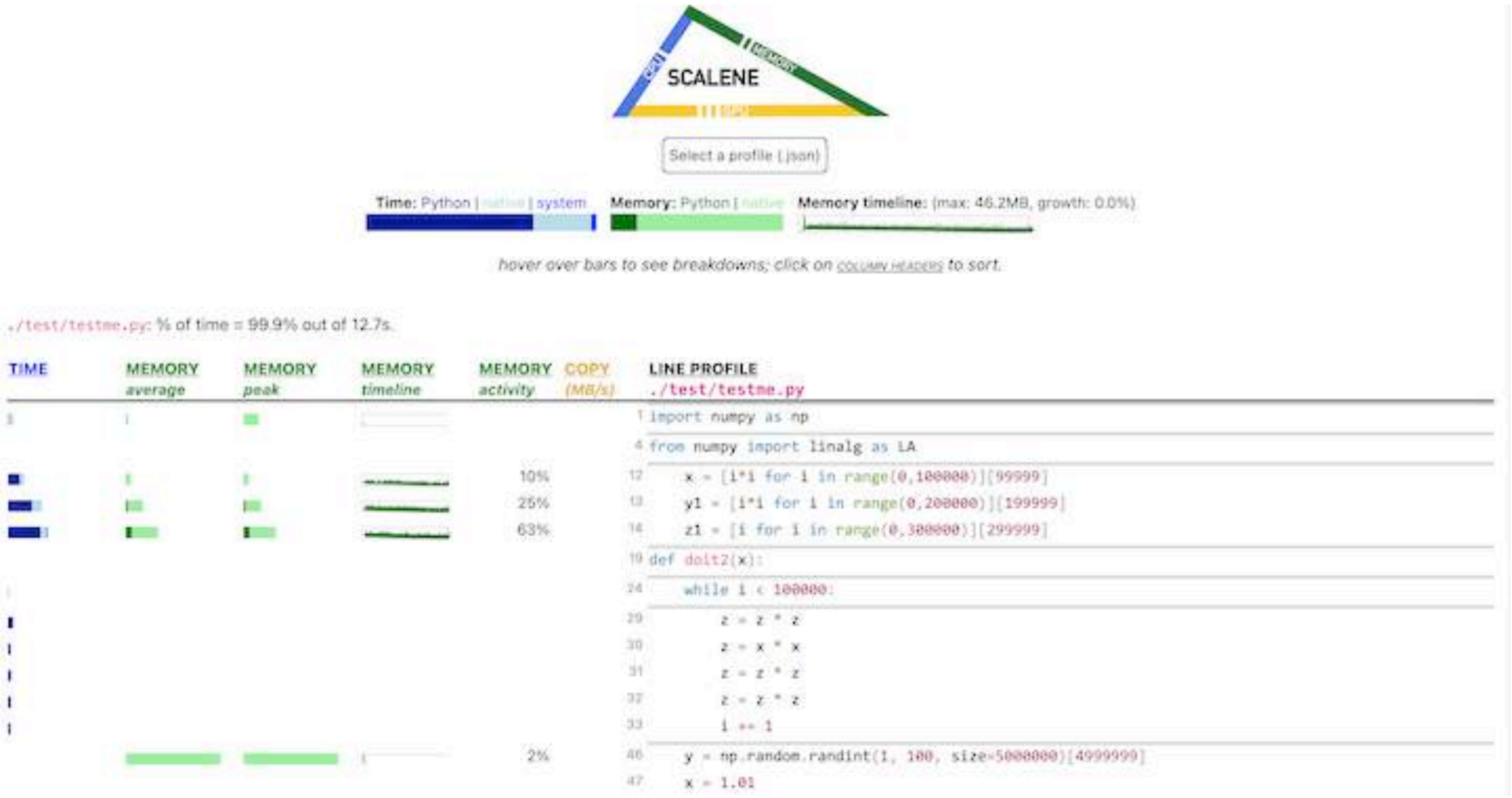
- Consider the performance/size of your storage when loading the data
- Use better deserialization primitives when available
- Try to avoid obvious inefficiencies when building task-specific pipelines

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 <u>parts</u> 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!



How to profile GPU code?

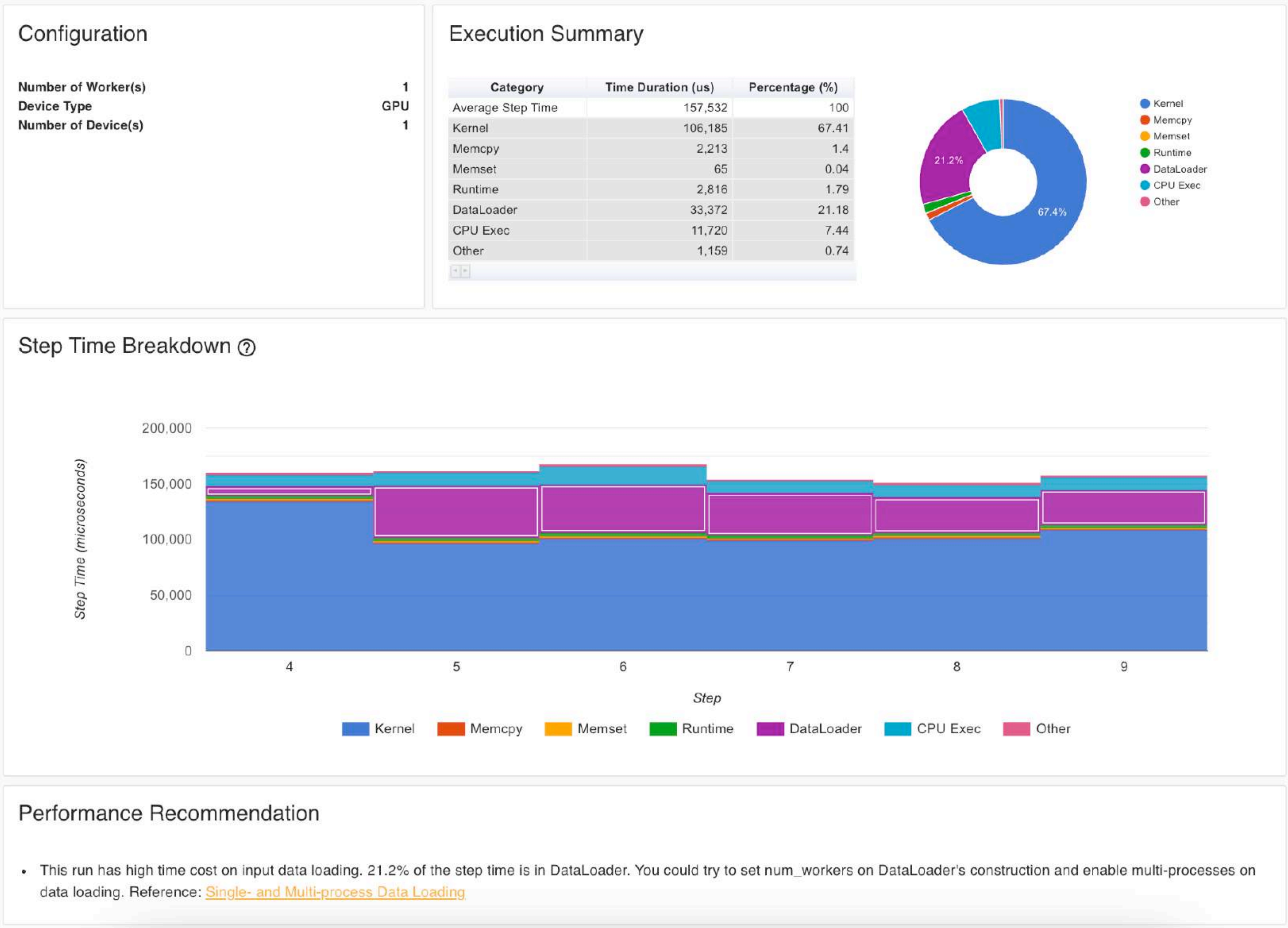
- nvprof is the low-level profiling tool
- Gives you the performance of low-level kernel launches and copies

```
==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]
```

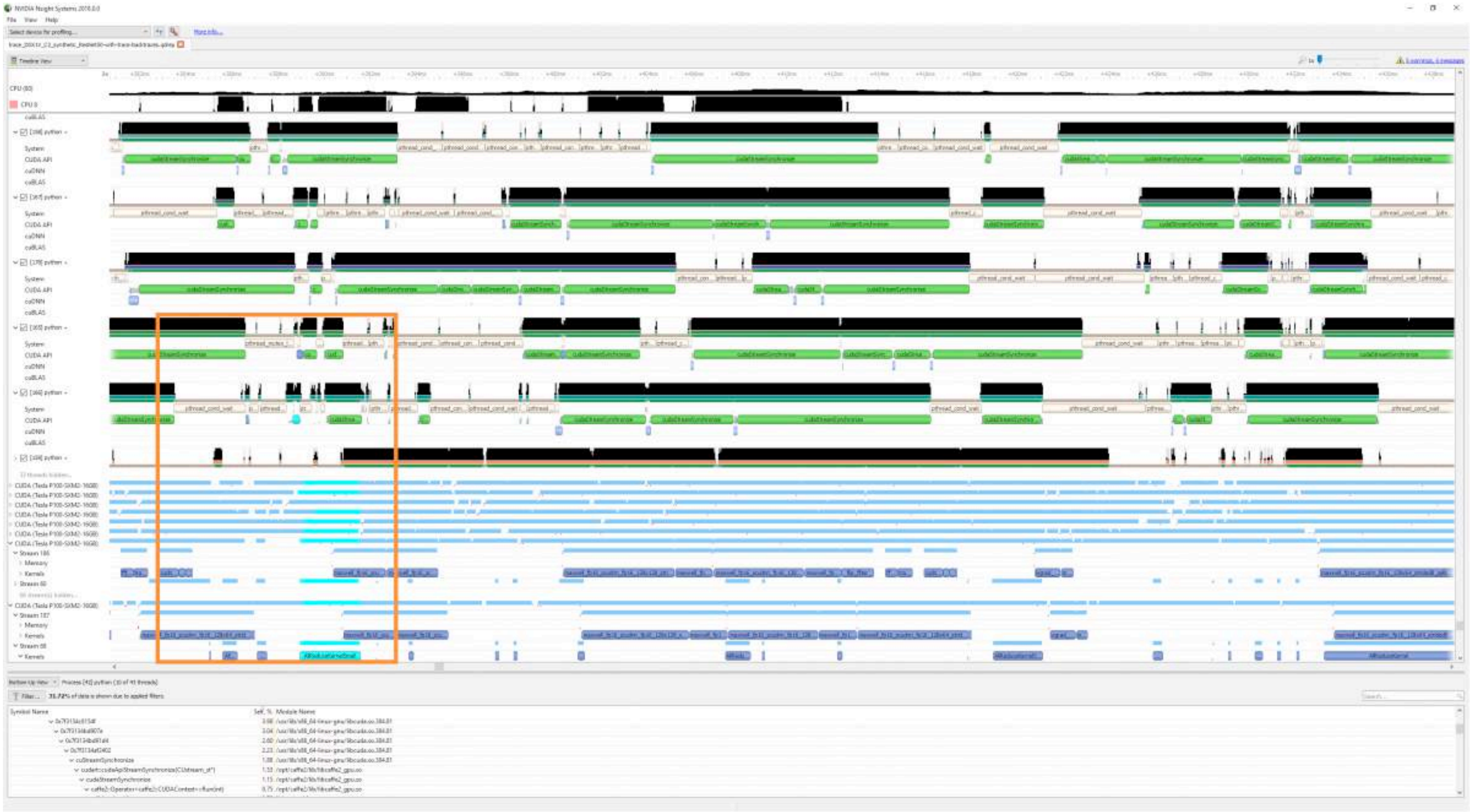
https://developer.nvidia.com/blog/cuda-pro-tip-nvprof-your-handy-universal-gpu-profiler/

How to profile PyTorch code?

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



Nsight Systems/Nsight Compute



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

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