



HPC for AI Research - Distributed Training

Andrea Pilzer, Solutions Architect | HPC for AI Research – ICIAP2025



Agenda

- Introduction

- TL;DR

- DDP

- Reduced Numerical Precision

- Dataloader

- ---
- ---

About me

Andrea Pilzer - apilzer@nvidia.com



- Since May 2022 Solution Architect at NVIDIA
 - SA HER, Leading NVAITC Italy
- Postdoc at Aalto (Helsinki, Finland)
 - Uncertainty quantification for deep learning
- Previous industrial experience (Huawei Ireland, Dublin)
 - Domain Adaptation
- Ph.D. in CS @ Trento (2016-2019)
 - Supervised by Elisa Ricci & Nicu Sebe
 - Main topic stereo matching

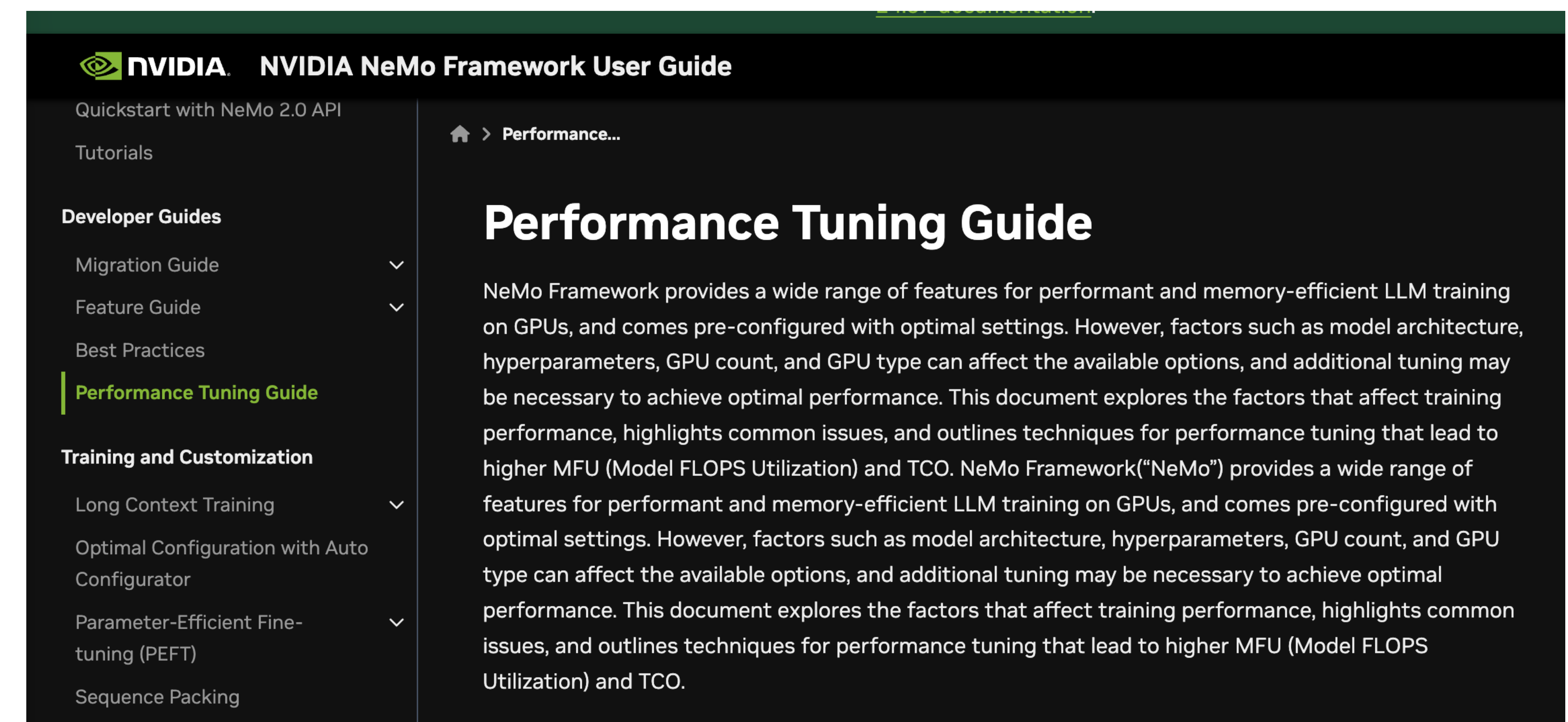


Introduction

Doing AI at Scale is Challenging

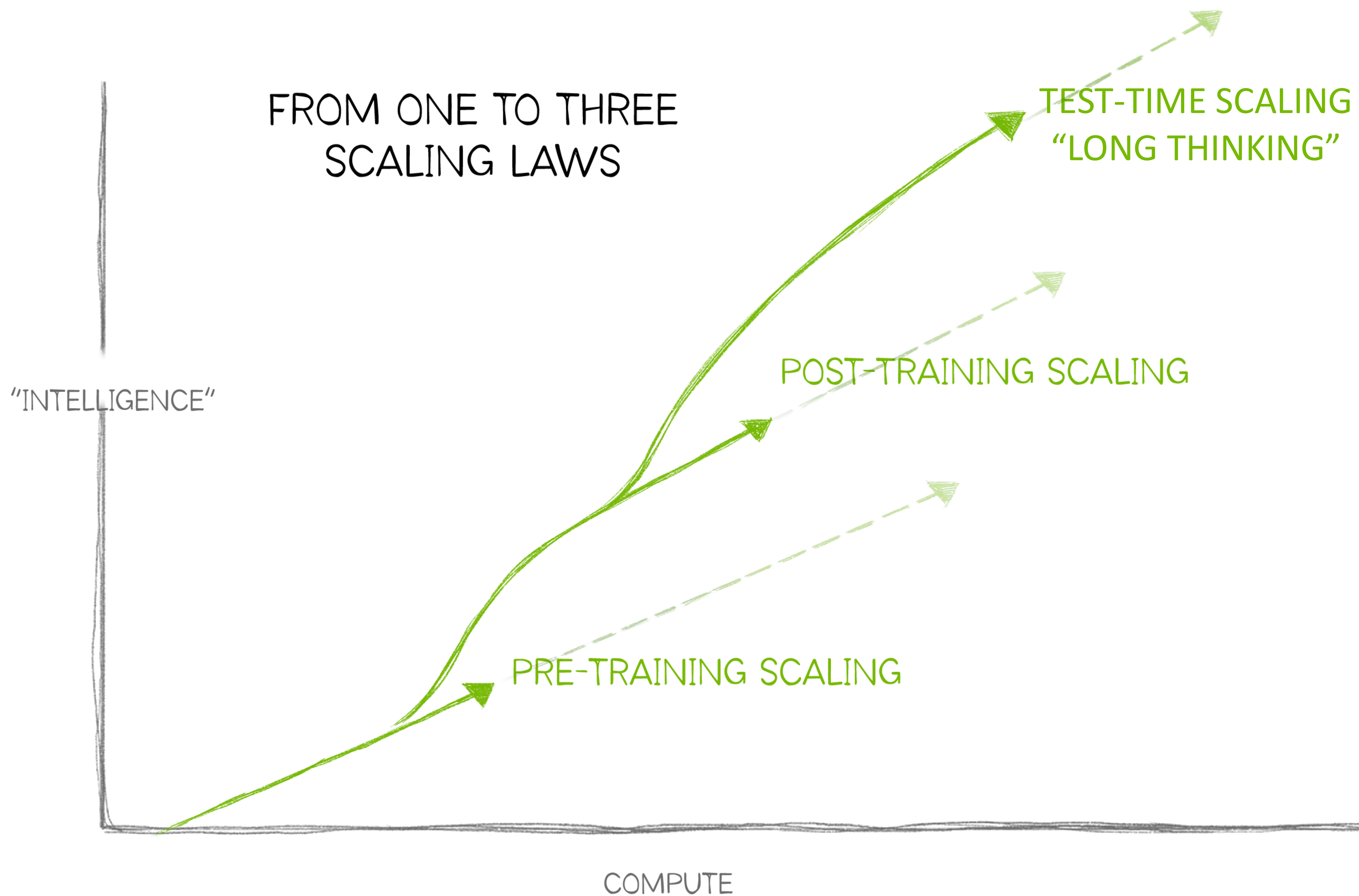
<https://docs.nvidia.com/nemo-framework/user-guide/latest/performance/performance-guide.html>

- The NeMo Performance Tuning Guide would be 25 pages if printed and is just a summary!
- Luckily HPC has been around for a long time and helps a lot
- In this talk we break down some of the techniques that allow us to run large scale experiments



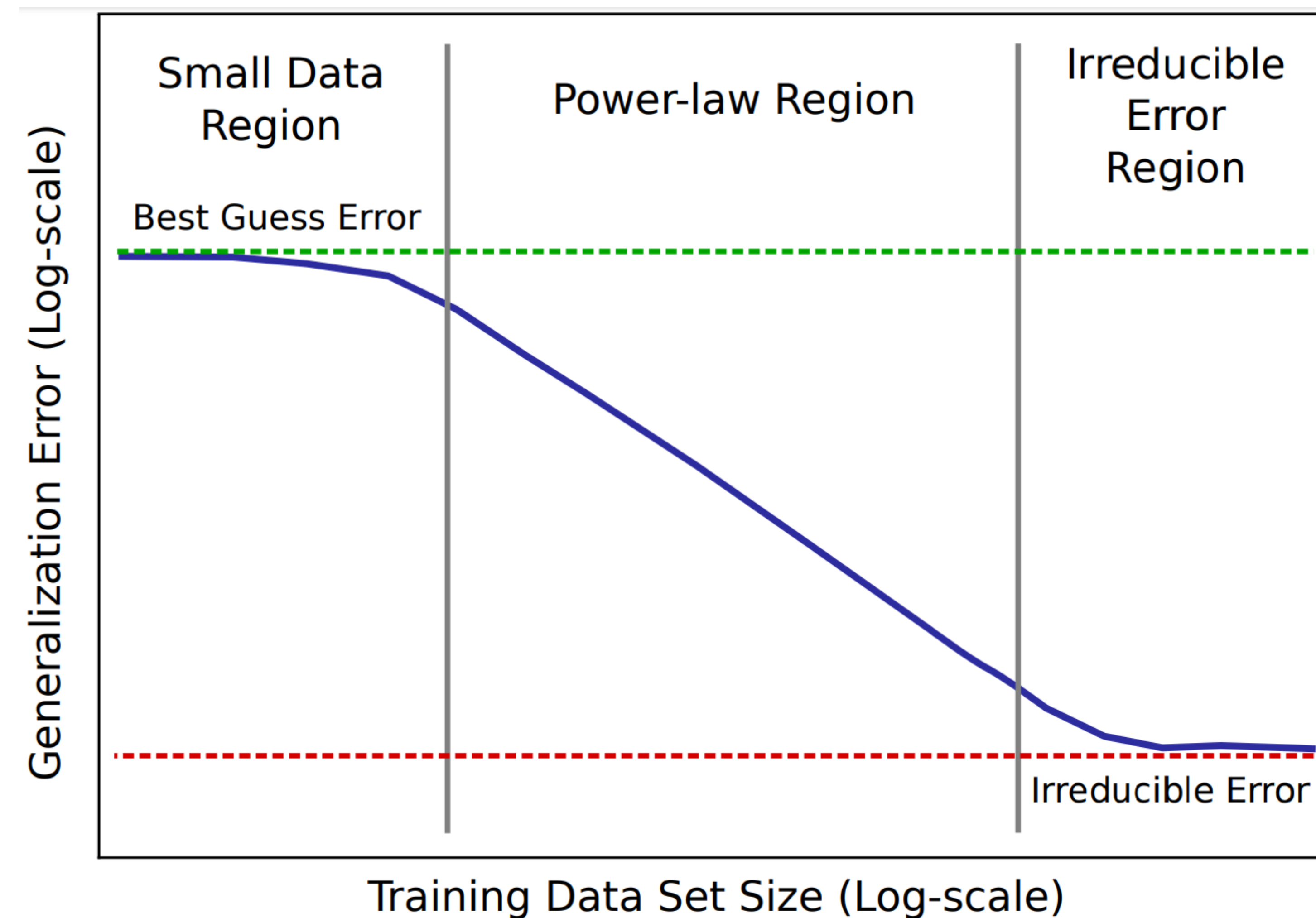
AI Scaling Laws Drive Exponential Demand for Compute

New “long thinking” supercharges inference scaling

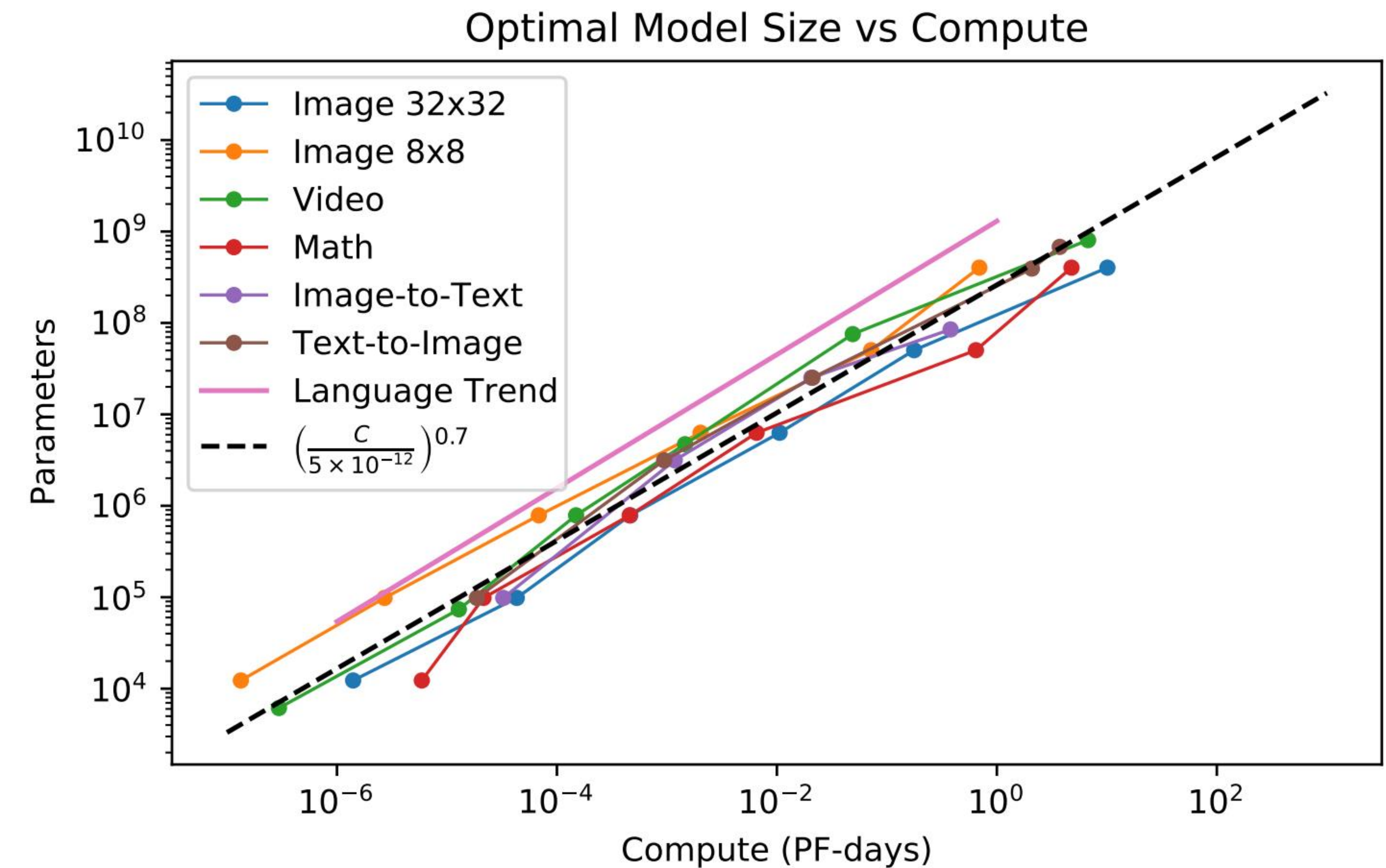


Scaling Laws of DL Training

Performance of neural networks increases with model/dataset size



Joel Hestness et al, Baidu Research, 2017
Deep Learning Scaling is Predictable, Empirically
arXiv:1712.00409



Tom Henighan et al, OpenAI, 2020
Scaling laws for autoregressive generative modeling
arXiv:2010.14701



TL;DR

This Session in One Slide (1)

Speed Memory Trade-Off

Single GPU

Method	Speed	Memory
Gradient Accumulation	No	Yes
Gradient Checkpointing	No	Yes
Mixed Precision Training	Yes	(No)
Batch Size	Yes	Yes
Optimizer Choice	Yes	Yes
DataLoader	Yes	No
Deepspeed Zero*	No	Yes

Multiple GPUs (4D parallelism)

- DP, Data Parallelism
- PP, Pipeline Parallelism (Model Parallelism)
- TP, Tensor Parallelism (Model Parallelism)
- SP, Sequence Parallelism (Activation Parallelism)
- CP, Context Parallelism (Activation Parallelism)

[\[1\] Efficient Training on a Single GPU](#)

(*) Listed here because it affects single GPU, but it is used for multi-GPU training

[\[2\] https://docs.nvidia.com/nemo-framework/user-guide/latest/nemotoolkit/features/parallelisms.html](https://docs.nvidia.com/nemo-framework/user-guide/latest/nemotoolkit/features/parallelisms.html)

[\[3\] Efficient Training on Multiple GPUs](#)


This Session in One Slide (2)

What do I use when? A super simplified take on it

- AI is a fine balance between networking and computing
- So which techniques should I use and when?
- First, optimize on one GPU
 - use memory friendly data formats
 - Check your dataloader hyperparameters
 - Use reduced precision
- After that, it depends, but generally these help 😊
 - DDP is the fastest if your model fits in memory
 - DDP + Zero/FSDP to reduce memory use
 - They might not be enough, for transformers we can use also TP typically within the same node to take advantage of NVLink interconnect for matrices sync
 - When they are not enough, we can play with PP
 - On top of PP, we can play with other techniques like offloading and activation recomputation, sometimes it is worth to recompute to save memory while anyway you wait for communications and synchronizations to happen

Notes:

In case you use NVIDIA Superchips like GH or GB offloading is very powerful thanks to the high CPU-GPU bandwidth



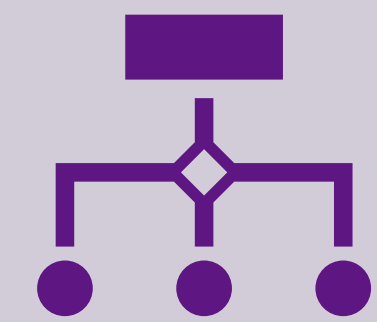
Distributed Data Parallelism (DDP)

(Distributed) Data Parallelism

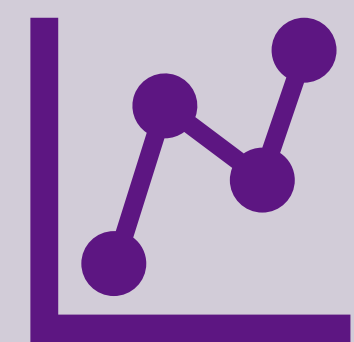
DDP vs DP



Distributed Data Parallelism fixes a weakness of DP, where one process controls all the GPUs training on a node



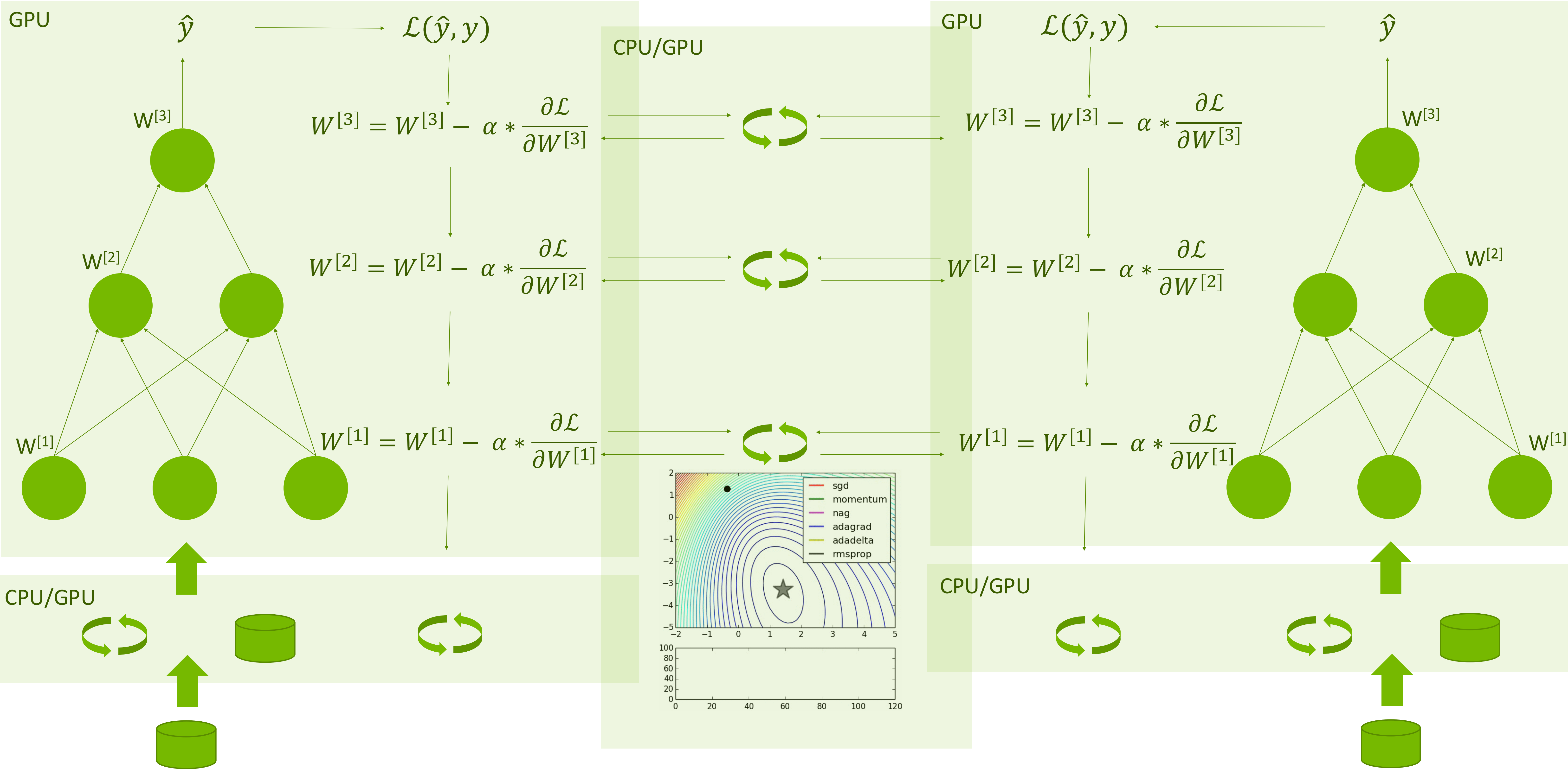
With DDP each GPU has its own task, they need to wait for each other only when synchronizing the gradients



There is also the option of using more advanced data parallelism techniques like Zero or FSDP but fundamentally the way your training works is the same

Training a Neural Network

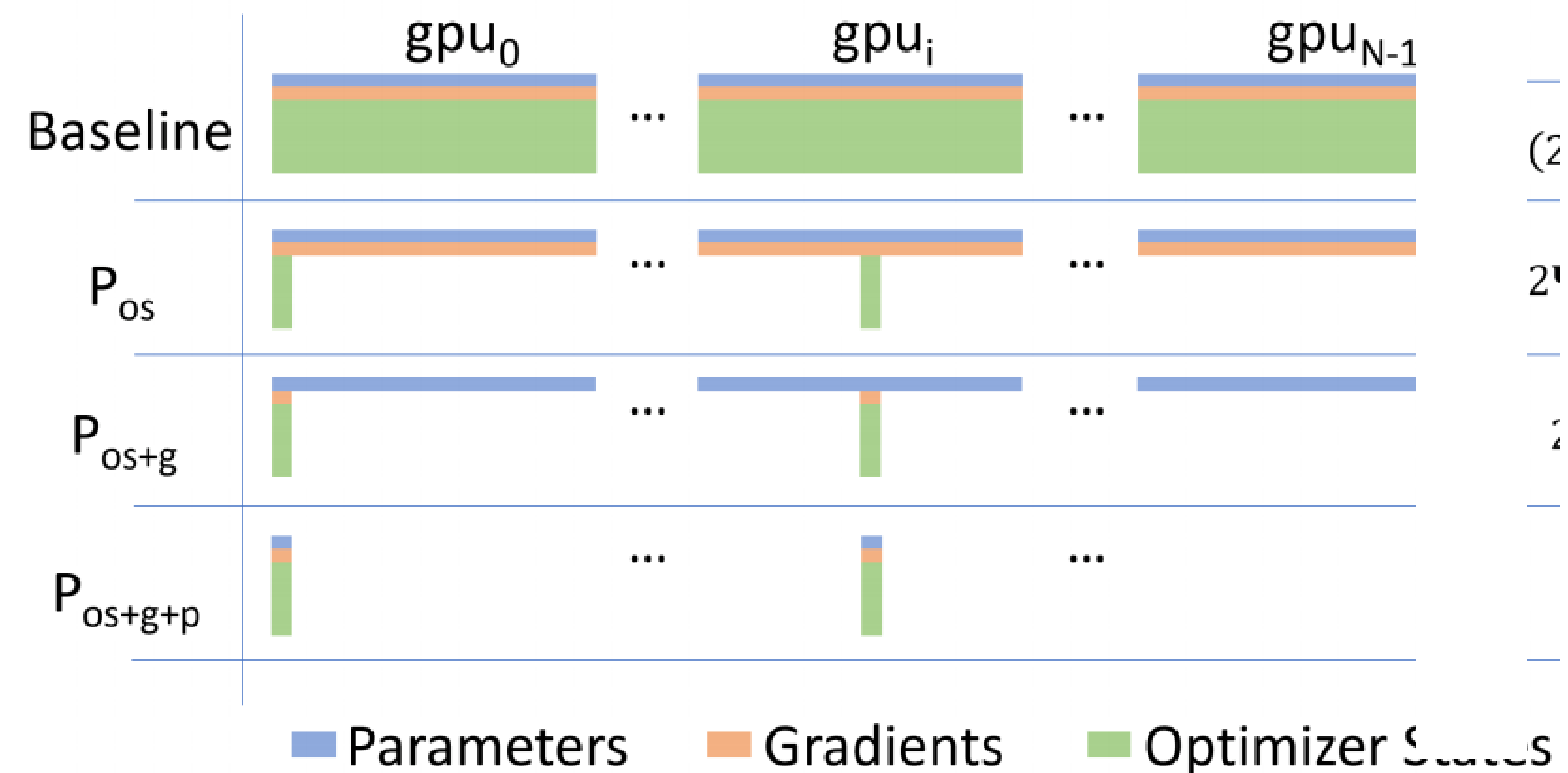
Multiple GPUs



Sharded Data Parallelism

ZeRO: Zero Redundancy Optimizer

- ZeRO removes the redundancy across data parallel process
- Partitioning optimizer states, gradients and parameters (3 stages) for a progressive memory savings and Communication Volume



Code (DDP),(Slurm)



Reducing Numerical Precision

Leonardo & Lisa @ CINECA

{BF16 vs FP8} or better {BF16 and FP8}

CINECA Tier 0 (=top) Cluster at the moment is **Leonardo**

- 4x NVIDIA A100 GPUs per node
- A100 Tensor Cores (Gen. 3) support FP16 HW accelerated GEMM

Later this year the extension **Lisa** will be installed

- 8x NVIDIA H100 GPUs per node
- H100 Tensor Cores (Gen. 4) support FP8 HW accelerated GEMM

In the following slides I'll introduce a bit about FP8 and AMP for Lisa and Leonardo.

TFLOPS

50x



GPU	FP32
Fermi (2010)	1.6
Pascal (2016)	10.6
Volta (2017)	14.9
Ampere (2020)	19.5
Hopper (2022)	66.9
Blackwell (2024)	80.0

Numerics contribute to speedup

50x

GPU	FP32	TF32	FP16	BF16	FP8	INT8	FP6	FP4
Fermi (2010)	1.6							
Pascal (2016)	10.6		21					
Volta (2017)	14.9		119					
Ampere (2020)	19.5	156	312	312		624		
Hopper (2022)	66.9	535	989	989	1,979			
Blackwell (2024)	80.0	1,110	2,250	2,250	4,500	4,500	4,500	9,000

112.5x

Benefits of less bits

Memory

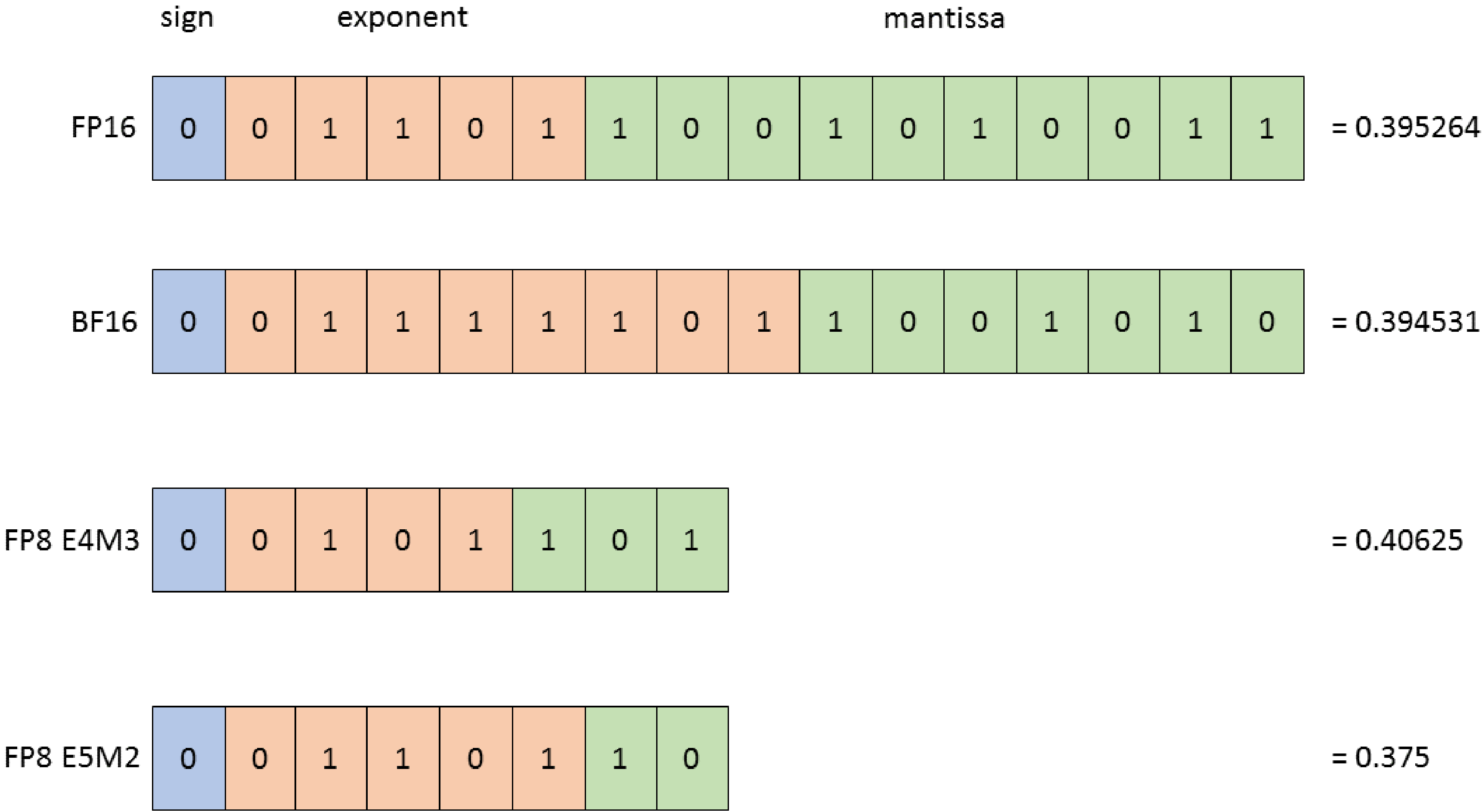
Weights and tensors occupy less space in memory

Bandwidth

Faster data movement from main memory
HBM to cores (and vice versa)

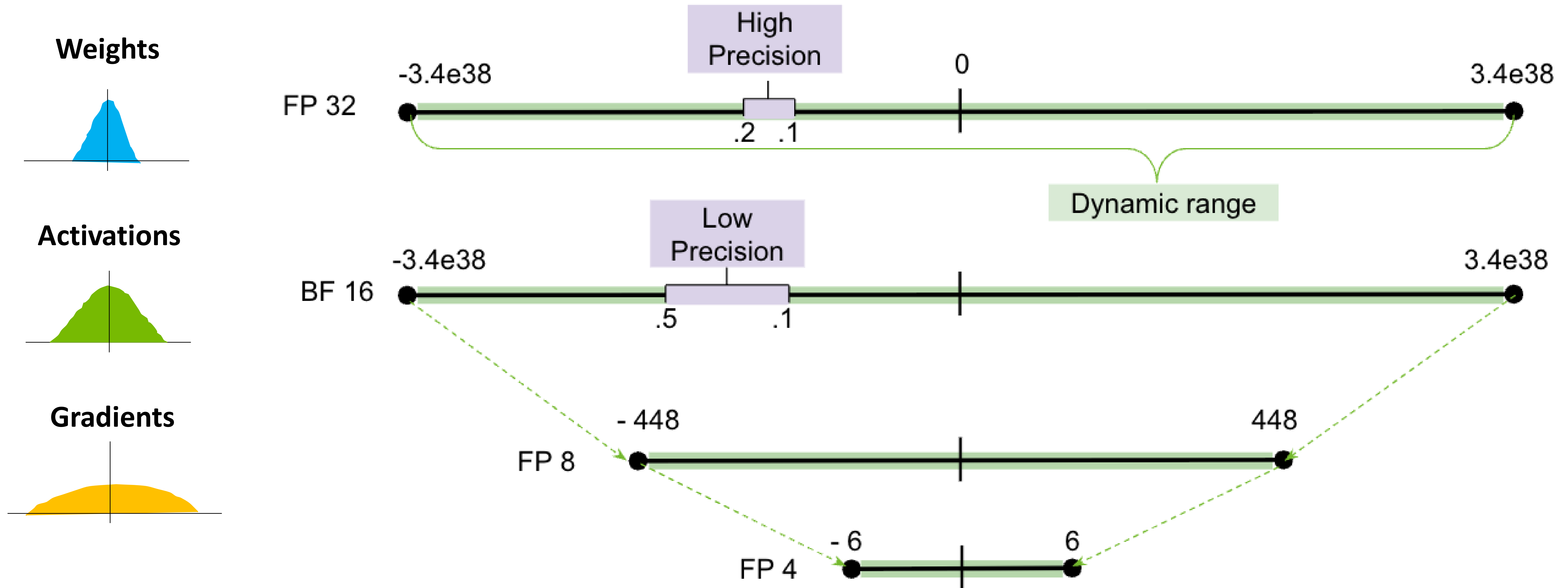
Compute

More TFLOPS with less bits
Faster matrix multiplications

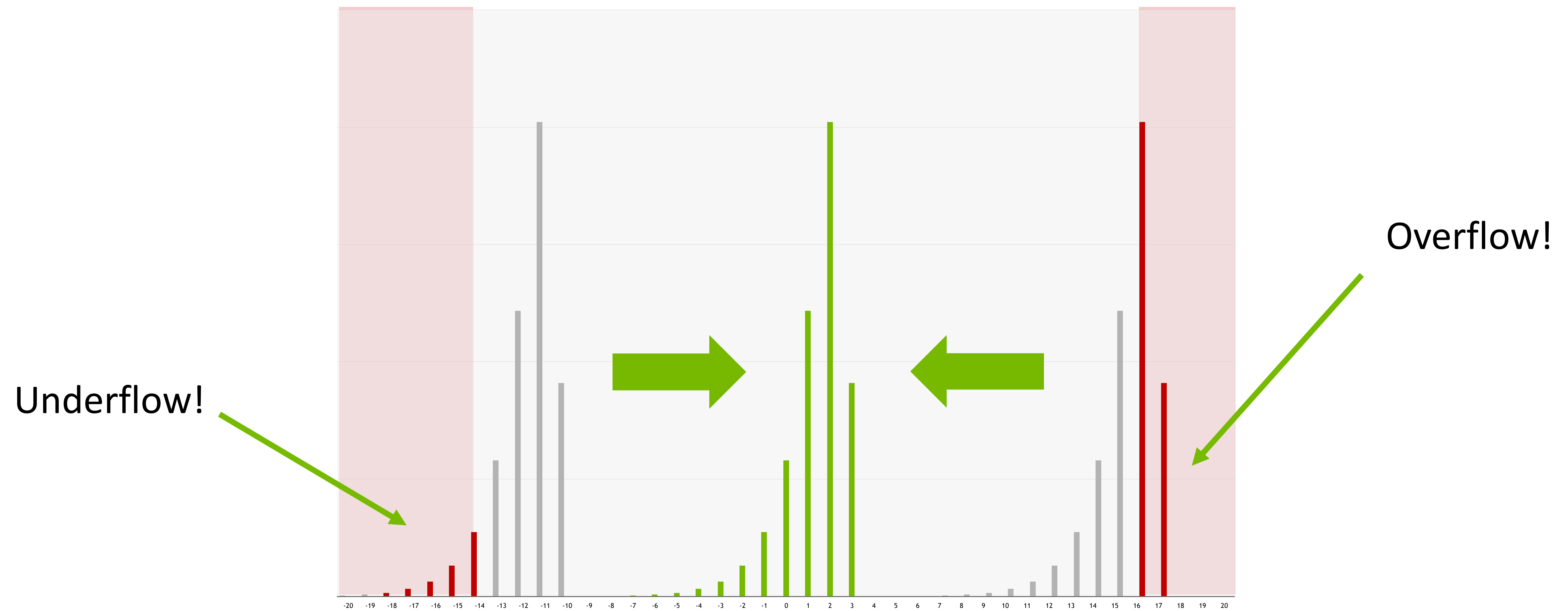


Wider distributions suffer more with Quantization

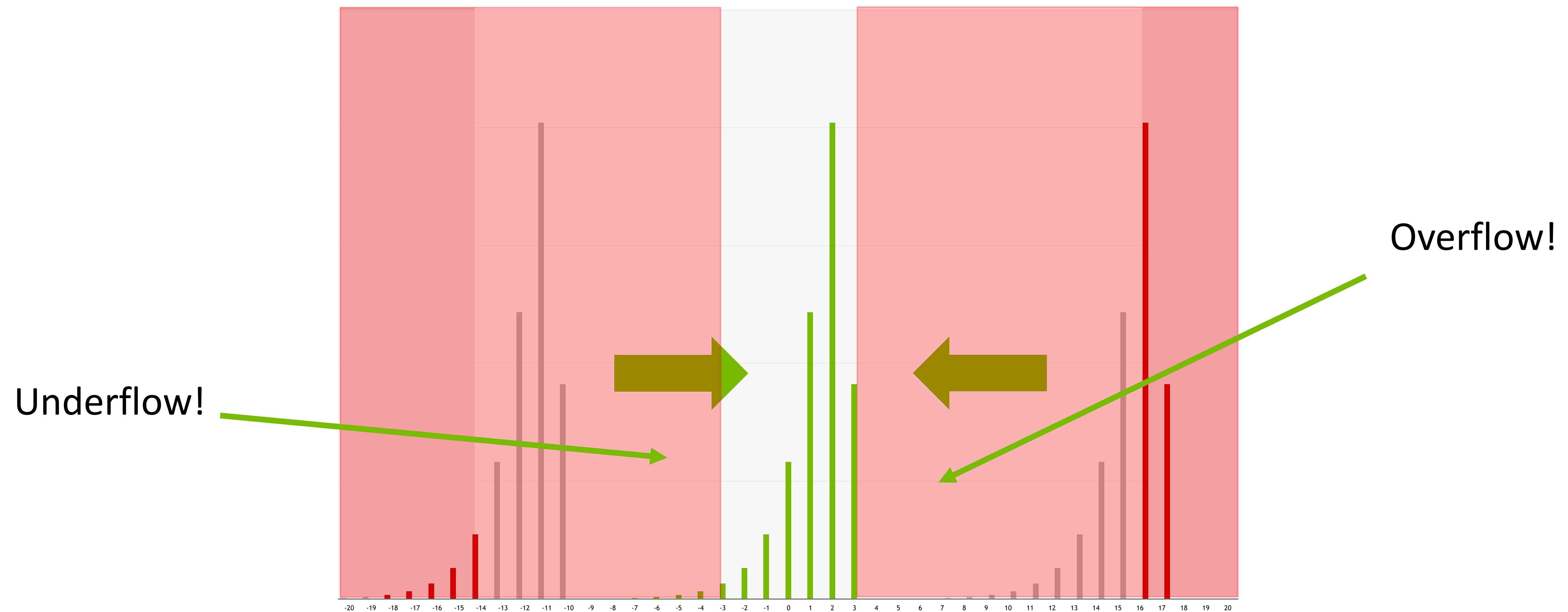
Gradients have the widest distributions



Scaling Factors to Keep Tensors within Range



Scaling Factors to Keep Tensors within Range



As we reduce the number of bits, the available range narrows

Long history of innovations to train in low-precision

Research, HW and SW are needed

Research

MIXED PRECISION TRAINING

Sharan Narang*, Gregory Diamos, Erich Elsen†
Baidu Research
{sharan, gdiamos}@baidu.com

Paulius Micike
Oleksii Kuchai
NVIDIA
{paulium,
okuchaiev,

FP8 FORMATS FOR DEEP LEARNING

Increasing
precision
reduces
training
time by
up to
10x

Paulius Micikevicius, Dusan Stosic, Patrick Judd, John Kamalu, Stuart Oberman, Mohammad Shoeybi,
Michael Siu, Hao Wu
NVIDIA
{paulium, dstosic, pjudd, jkamalu, soberman, mshoeybi, msui, skyw}@nvidia.com

Microscaling Data Formats for Deep Learning

{naveen.k.melle,
naveen.k.melle}

Bitar Darvish Rouhani*, Ritchie Zhao, Ankit More, Mathew Hall, Alireza Khodamoradi, Summer Deng,
Dhruv Choudhary, Marius Cornea, Eric Dellinger, Kristof Denolf, Stosic Dusan, Venmugil Elango,
Maximilian Golub, Alexander Heinecke, Phil James-Roxby, Dharmesh Jani, Gaurav Kolhe,
Martin Langhammer, Ada Li, Levi Melnick, Maral Mesmahhosroshahi, Andres Rodriguez,
Michael Schulte, Rasoul Shafipour, Lei Shao, Michael Siu, Pradeep Dubey, Paulius Micikevicius,
Maxim Naumov, Colin Verrilli, Ralph Wittig, Doug Burger, Eric Chung

Microsoft, AMD, Intel, Meta, NVIDIA, Qualcomm Technologies Inc.

Abstract

Narrow bit-width data formats are key to reducing the computational and storage costs of modern deep learning applications. This paper evaluates Microscaling (MX) data formats that combine a per-block scaling factor with narrow floating-point and integer types for individual elements. MX formats balance the competing needs of hardware efficiency, model accuracy, and user friction. Empirical results on over two dozen benchmarks demonstrate practicality of MX data formats as a drop-in replacement for baseline FP32 for AI inference and training with low user friction. We also show the first instance of training generative language models at sub-8-bit weights, activations, and gradients with minimal accuracy loss and no modifications to the training recipe.

Hardware

GPU	FP32	TF32	FP16	BF16	FP8	INT8	FP6	FP4
Fermi (2010)	1.6							
Pascal (2016)	10.6		21					
Volta (2017)	14.9		119					
Ampere (2020)	19.5	156	312	312		624		
Hopper (2022)	66.9	535	989	989	1,979	1,979		
Blackwell (2024)	80.0	1,100	2,250	2,250	4,500	4,500	4,500	9,000

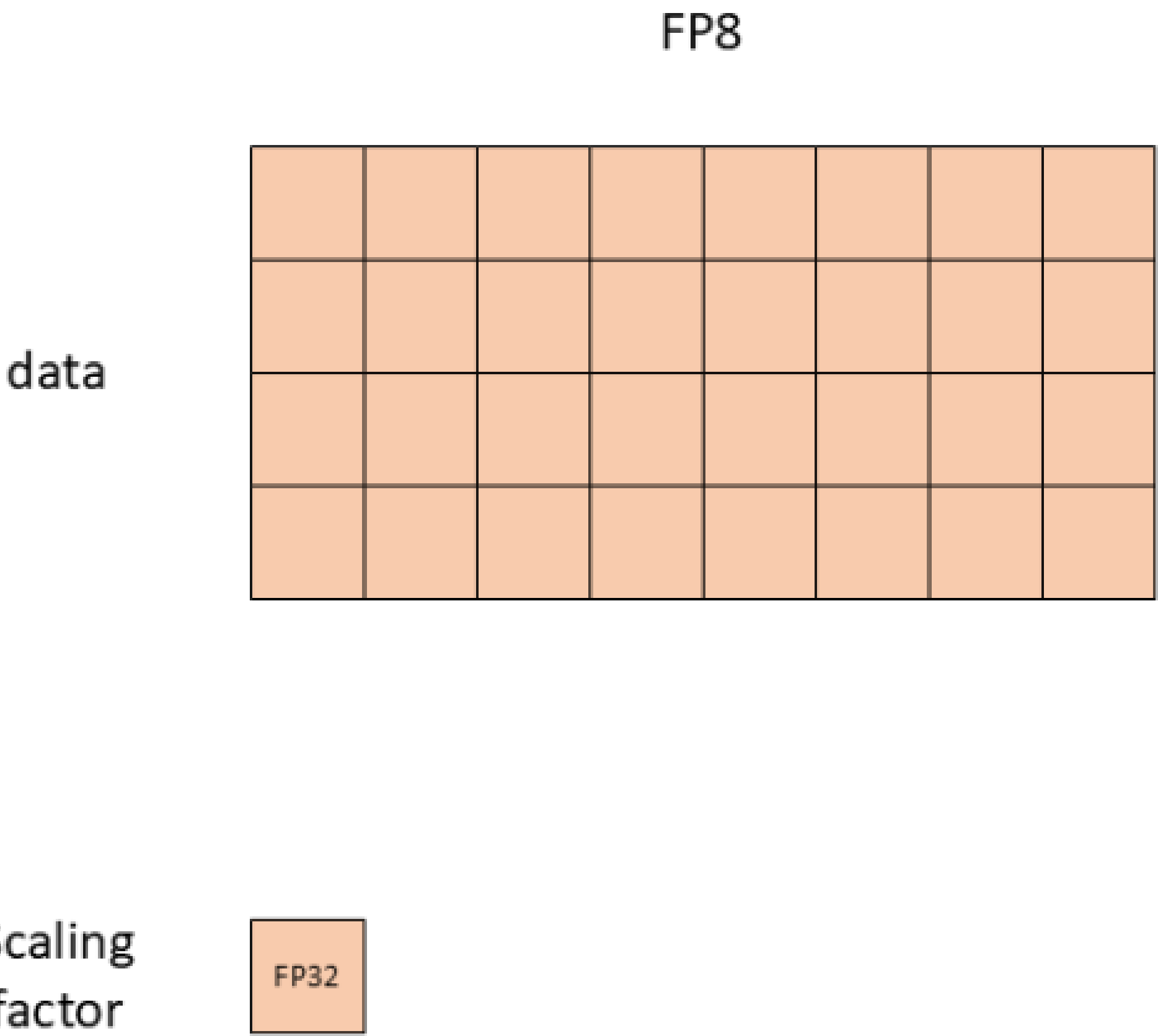
Software



Towards finer Scaling Factors

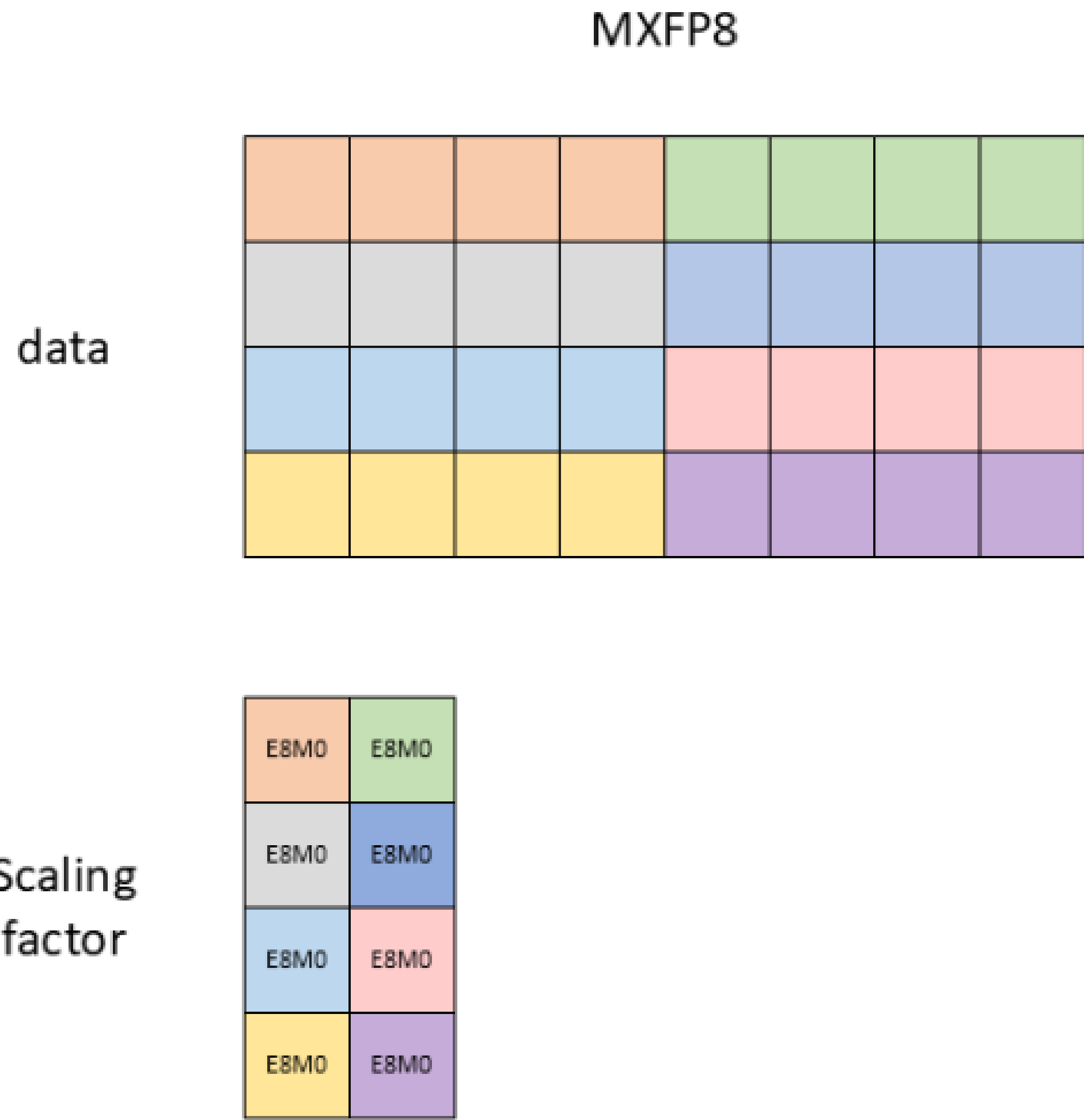
Ampere

FP16 mixed precision
one scale for all tensors



Hopper

FP8 training
one scale per tensor

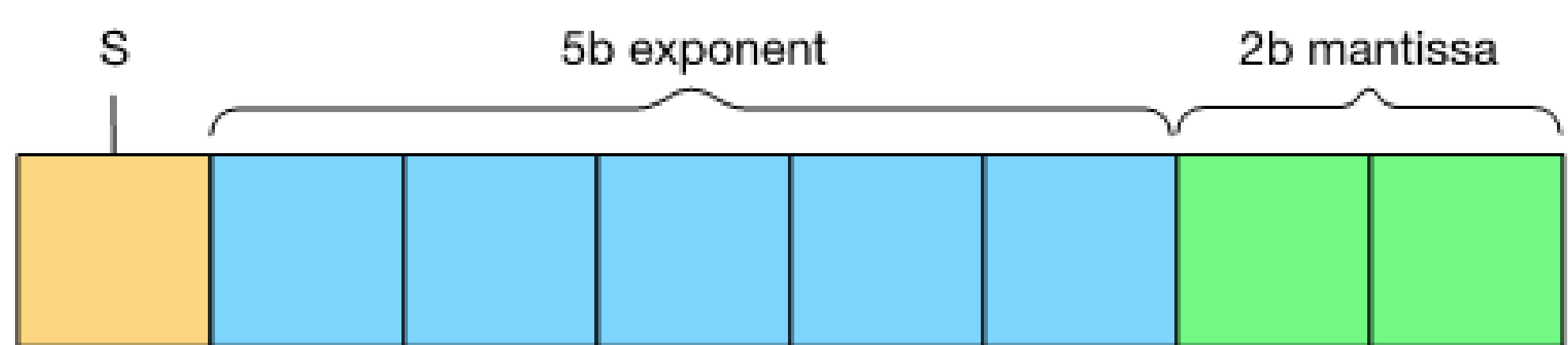
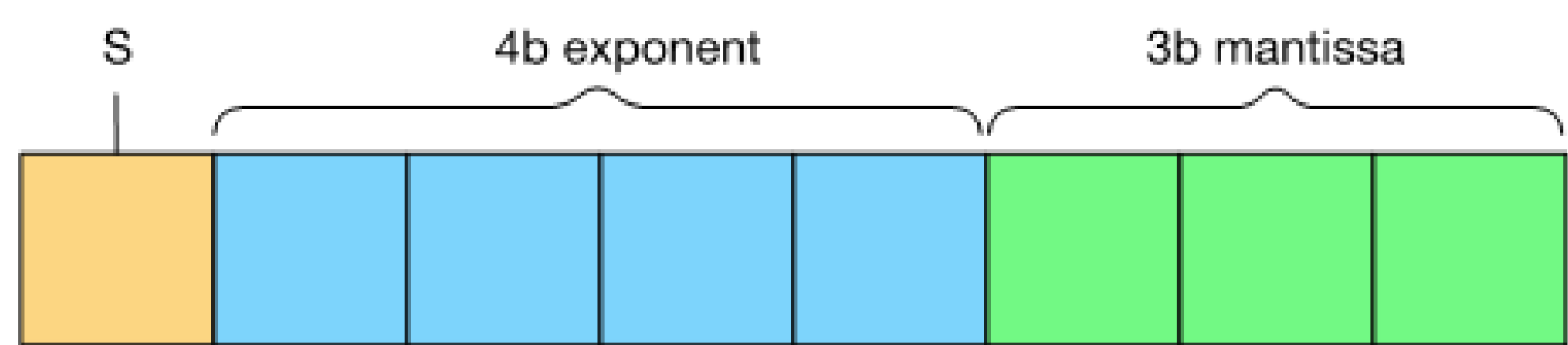


Blackwell

MXFP8 training*
one scale per block of 32 elements

*MX stands for microscaled formats, read [OCP spec](#)

FP8 Data Type



	E4M3	E5M2
Exponent Bias	7	15
Infinities	N/A	$S.11111.00_2$
NaNs	$S.1111.111_2$	$S.11111.\{01, 10, 11\}_2$
Zeros	$S.0000.000_2$	$S.00000.00_2$
Max normal	$S.1111.110_2 = 1.75 * 2^8 = 448$	$S.11110.11_2 = 1.75 * 2^{15} = 57344$
Min normal	$S.0001.000_2 = 2^{-6}$	$S.00001.00_2 = 2^{-14}$
Max subnormal	$S.0000.111_2 = 0.875 * 2^{-6}$	$S.00000.11_2 = 0.75 * 2^{-14}$
Min subnormal	$S.0000.001_2 = 2^{-9}$	$S.00000.01_2 = 2^{-16}$
Dynamic range	~18 binades	~32 binades

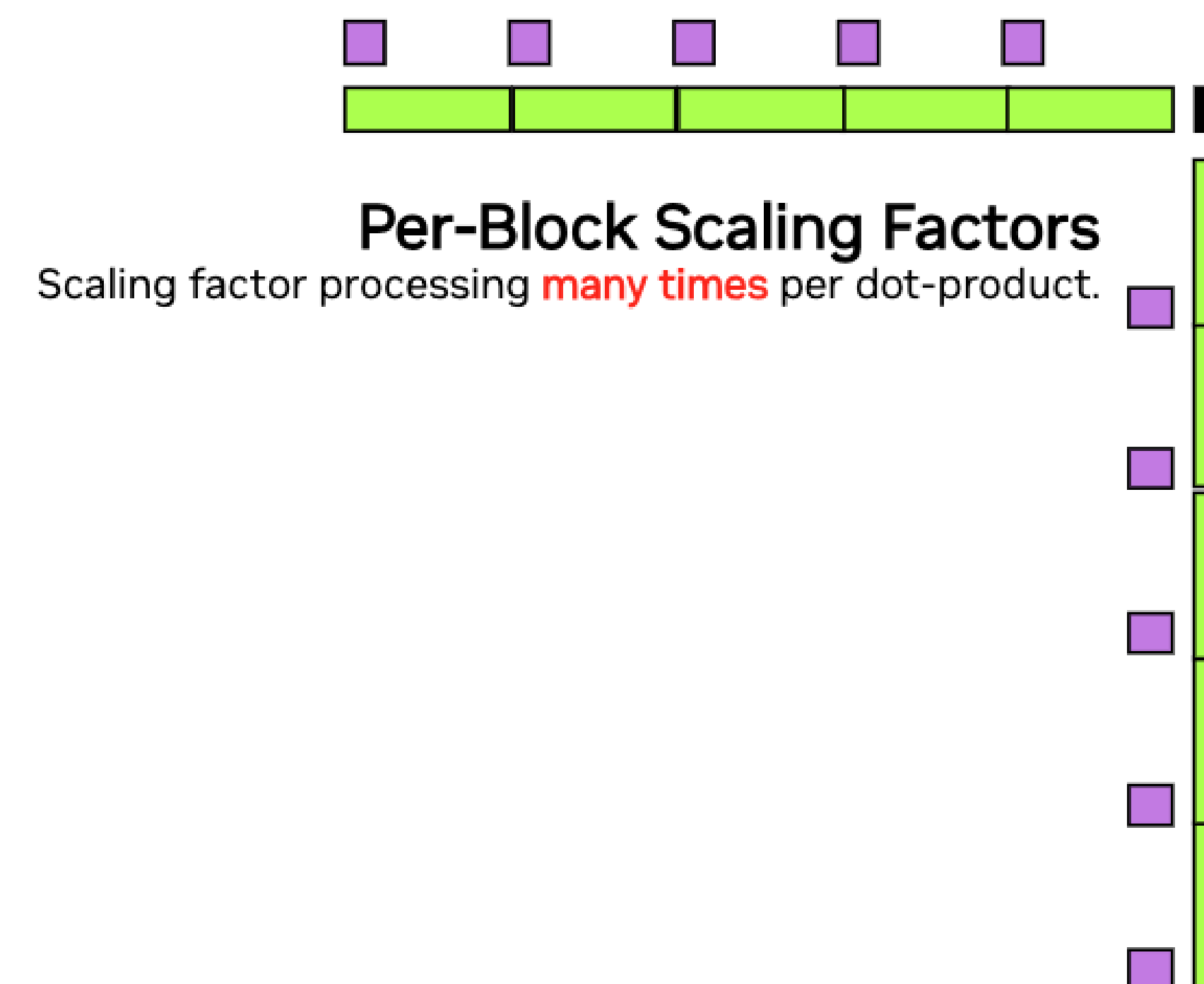
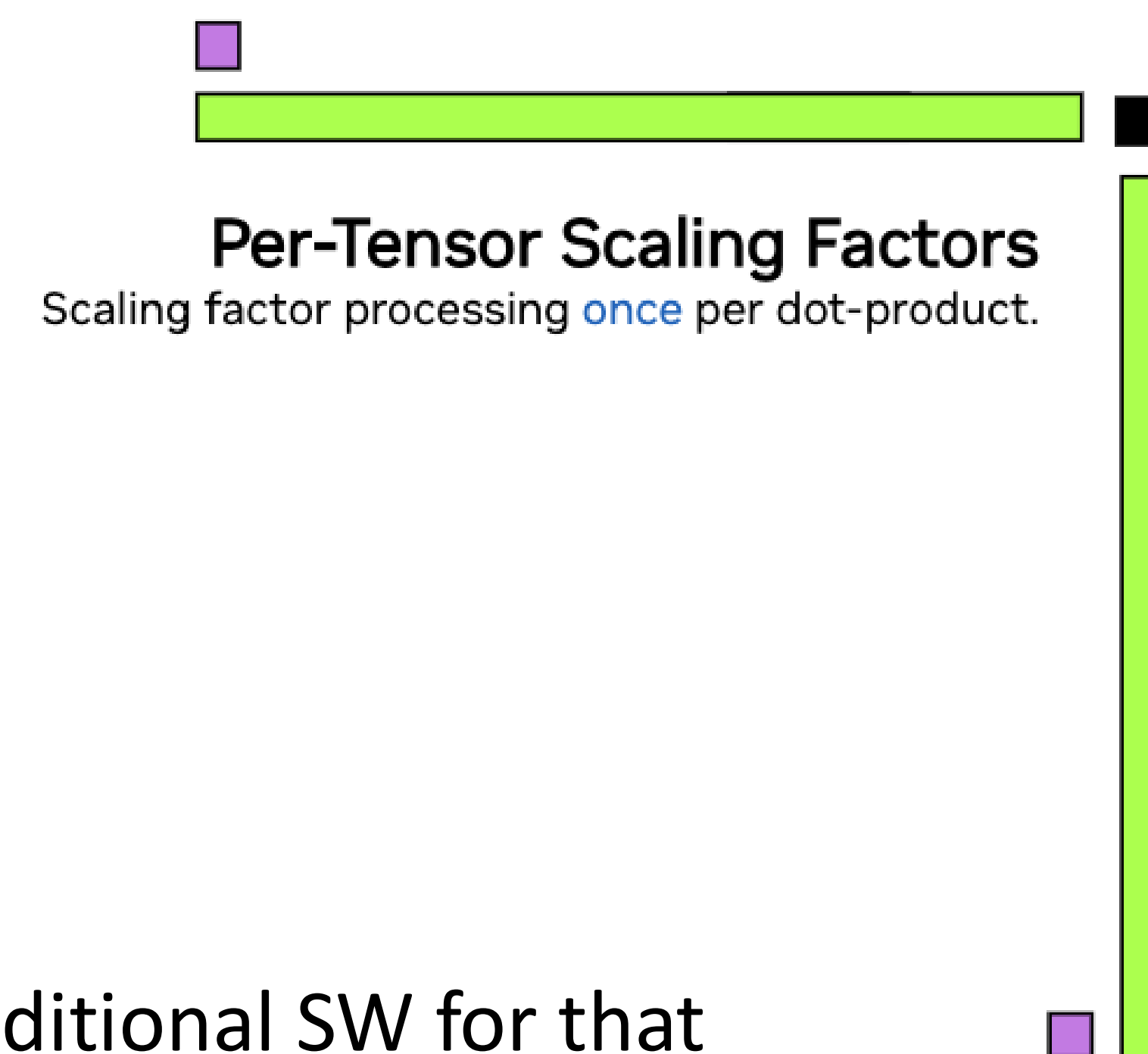
E4M3 is the only FP8 format used in MXFP8 training recipe

Microscaled formats need of special hardware instructions

Doing it in software is expensive

- Think of a **dot-product of large vectors**: fundamental op in LLMs
- Scaling factors (metadata) need processing, once per group of values that share that metadata:
 - **Per-tensor scaling factor**: once at the end of the dot-product, after all the Tensor Core ops (i.e. cheap to do in SW)
 - **Per-block scaling factor**: once after each block – many times per dot-product (expensive to do in SW)

fundamental



Tensor Core, not requiring additional SW for that

- **5th Gen Tensor Cores** (Blackwell):
 - Take data and metadata as inputs to *MMA instructions
 - Handle scaling-factor

FP8 Recipes Available



[Hopper] Current Scaling + 1st & last layer BF16

- Current scaling replacing delayed scaling as best known per-tensor recipe
- Keeps the more sensitive 1st and last layer in BF16
- E4M3 for weights and activations, E5M2 for gradients

[Hopper] NV Subchannel Recipe (DeepSeek-V3 like)

- As DeepSeek-V3 pretraining
- 1x128 blocks for input and output_grad, 128x128 blocks for weights
- E4M3 for all weights, acts, and grads

[Blackwell] MXFP8 Blockwise Scaling

- Different scaling factor for each block of 32 values in a tensor
- E4M3 for all weights, acts, and grads

Transformer Engine

- An open-source library implementing the FP8 recipe for Transformer building blocks
- Optimized for FP8 and other datatypes
- PyTorch and JAX are supported frameworks.
- Composable with the native framework operators.
- Supports different types of model parallelism
 - DP, TP, PP, CP
- <https://github.com/NVIDIA/TransformerEngine>
- Docs:
 - <https://docs.nvidia.com/deeplearning/transformer-engine/user-guide/index.html>

```
import torch
import transformer_engine.pytorch as te
from transformer_engine.common import recipe

# Set dimensions.
in_features = 768
out_features = 3072
hidden_size = 2048

# Initialize model and inputs.
model = te.Linear(in_features, out_features, bias=True)
inp = torch.randn(hidden_size, in_features, device="cuda")

# Create MXFP8 recipe.
fp8_recipe = recipe.MXFP8BlockScaling()

# Enable autocasting to FP8.
with te.fp8_autocast(enabled=True, fp8_recipe=fp8_recipe):
    out = model(inp)

# Calculate loss and gradients.
loss = out.sum()
loss.backward()
```

Recipes Available in NeMo FW and Megatron-LM

The backend is Transformer Engine

- [NVIDIA NeMo framework](#) allows developers to easily run multi-node training of LLMs
- [Megatron-LM](#) is research-oriented framework to train LLMs
- Recipes available in https://github.com/NVIDIA/NeMo/blob/main/nemo/collections/llm/recipes/precision/mixed_precision.py

```
trainer = nl.Trainer(  
    devices=args.devices,  
    num_nodes=args.num_nodes,  
    max_steps=args.max_steps,  
    log_every_n_steps=args.log_interval,  
    val_check_interval=args.val_check_interval,  
    limit_val_batches=args.limit_val_batches,  
    strategy=strategy,  
    accelerator="gpu",  
    plugins=fp16_with_mxfp8_mixed(), ←  
    use_distributed_sampler=False,  
    callbacks=[itr_rate_callback],  
)
```

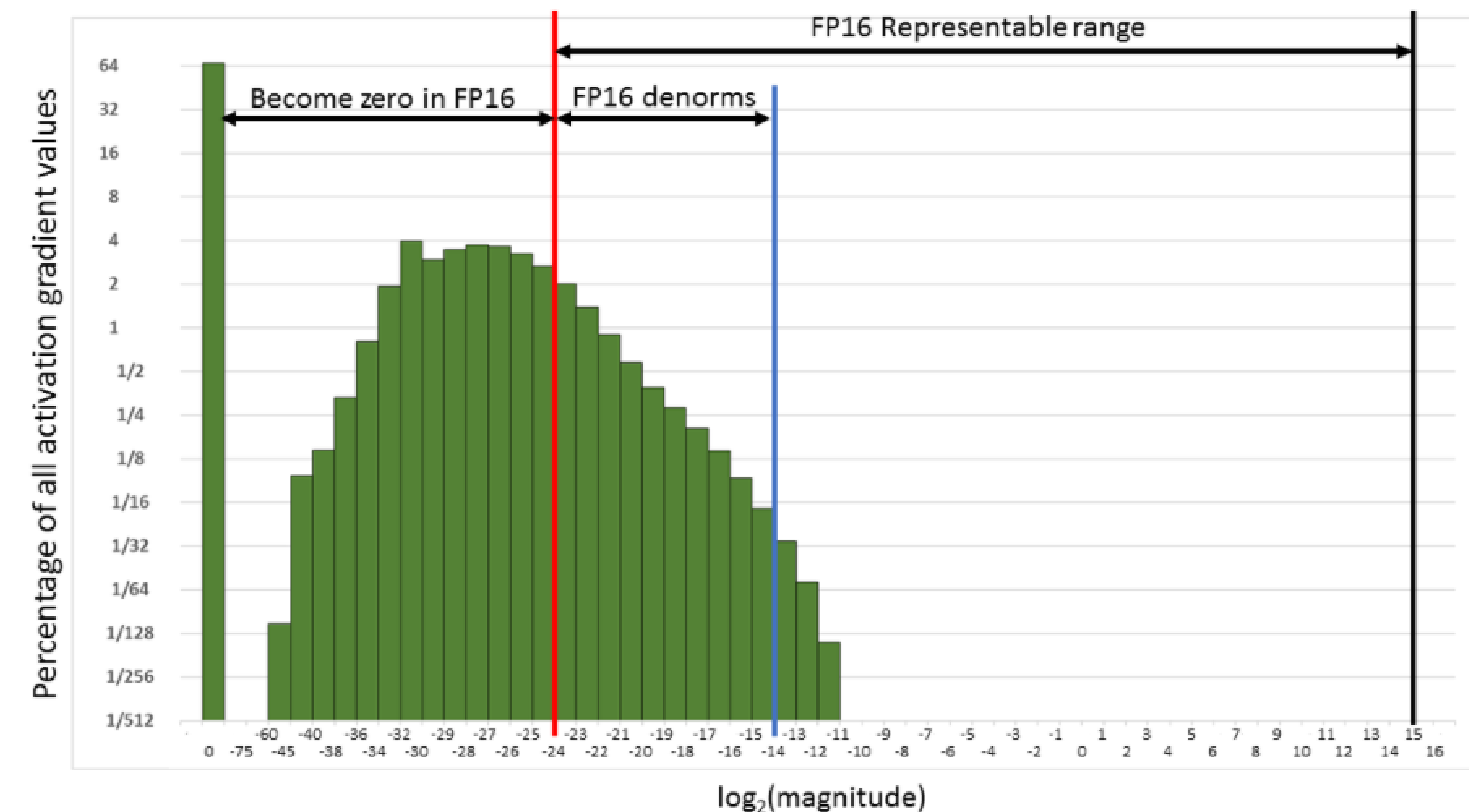
```
def fp16_with_mxfp8_mixed() -> run.Config[MegatronMixedPrecision]:  
    """Create a MegatronMixedPrecision plugin configuration for mixed  
  
    Returns:  
        run.Config[MegatronMixedPrecision]: Configuration for FP16 with  
        """  
    cfg = fp16_mixed()  
    cfg.fp8 = 'hybrid'  
    cfg.fp8_recipe = "mxfp8"  
    cfg.fp8_param_gather = False  
    return cfg
```




AMP is still an option

Mixed Precision Training

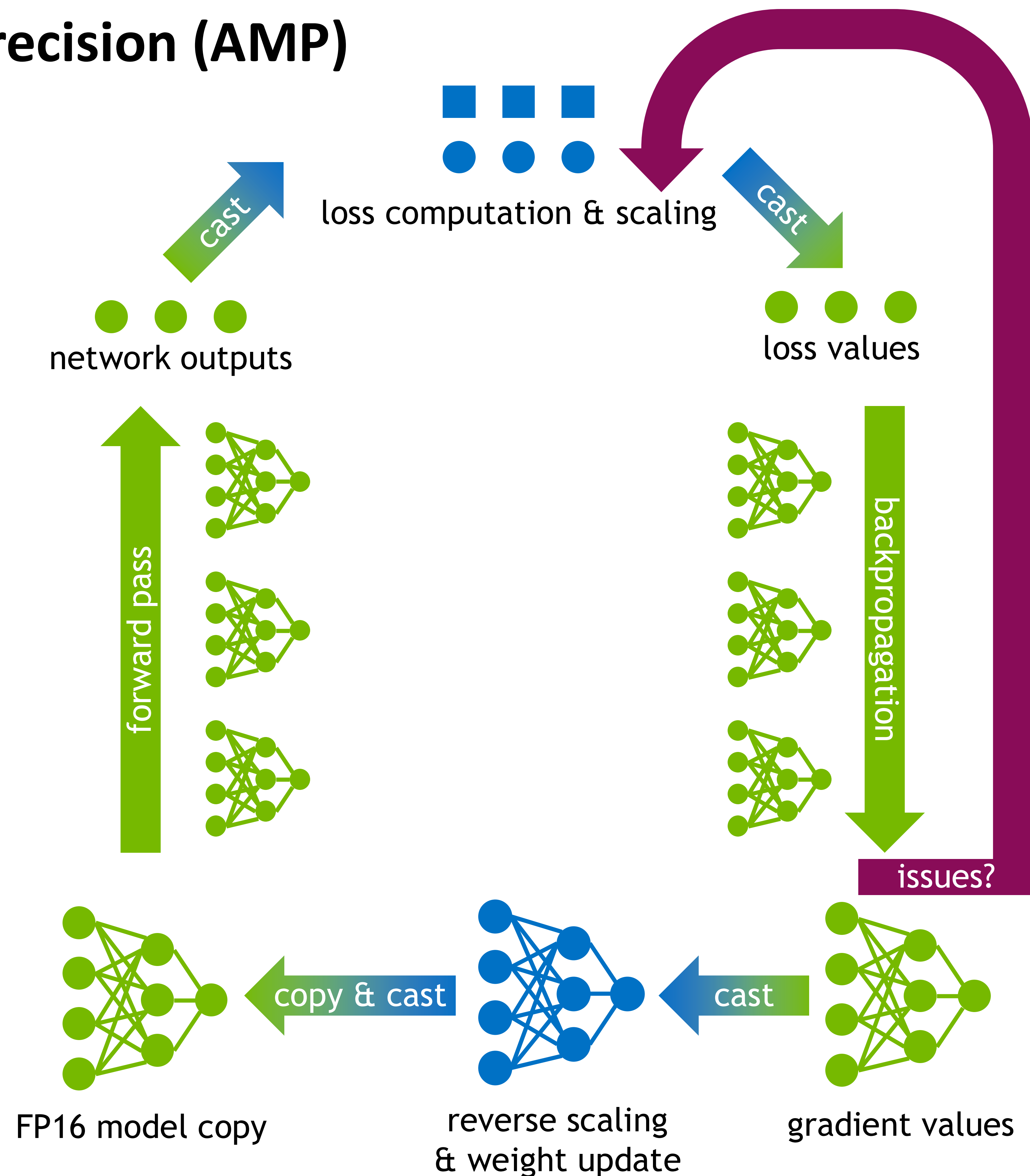
- Histogram shows activation gradient magnitudes throughout FP32 training; both axes are logarithmic.
- Observations:
 - Dynamic range of FP16 would be sufficient to cover the entire histogram. 😊
 - Without “shifting” the histogram, half of the activations would be casted to 0, however. 😞
- Idea: “shifting” = multiplication with a scale factor!
- Concern: Do I need to run a full training in order to find the scaling factor? → No, automatic mixed precision comes to the rescue! 😊



Automatic Mixed Precision (AMP)

Concept

- Maintain a primary copy of weights in FP32.
- Initialize scaling factor S to a large value.
- For each iteration:
 - Make an FP16 copy of the weights.
 - Forward propagation (FP16 weights and activations).
 - Multiply the resulting loss with the scaling factor S .
 - Backward propagation (FP16 weights, activations, and their gradients).
 - If there is an Inf or NaN in weight gradients:
 - Reduce S .
 - Skip the weight update and move to the next iteration.
 - Multiply the weight gradient with $1/S$.
 - Complete the weight update (including gradient clipping, etc.).
 - If there hasn't been an Inf or NaN in the last N iterations, increase S .



How to use It?

in pytorch

- Backward passes under autocast are not recommended.
- Backward ops run in the same dtype autocast chose for corresponding forward ops.
- `scaler.step()` first unscales the gradients of the optimizer's assigned params.
- If these gradients contain infs or NaNs, `optimizer.step()` is skipped.

```
# initialize gradient scaler
scaler = GradScaler()

# training loop
for epoch in epochs:
    for input, target in data:

        # zero gradient buffers
        optimizer.zero_grad()

        # forward pass with autocasting
        with autocast():
            output = model(input)
            loss = loss_fn(output, target)

        # call backward() on scaled loss
        scaler.scale(loss).backward()

        # update if no issues
        scaler.step(optimizer)

        # updates the scale for next iteration.
        scaler.update()
```


Code (AMP)



Dataloader

Dataloaders

- Think of the GPU as a very powerful parallel processing device hungry for data
- Dataloaders have very important parameters that you can tune
 - Workers, how many subprocesses the dataloader can create
 - Prefetching, how many batches each worker will load at the time
 - Pin memory, allow workers to always use a specific memory address in CPU & GPU
- Asynchronous copy, CUDA can help in hiding data moving cost behind other operations
 - `data = data.to(device) ⇒ data = data.to(device, non_blocking=True)`
- There are also specific libraries you can use to accelerate dataloading like NVIDIA DALI for images and PyNvVideoCodec for videos

NVIDIA Video/Image Processing Hardware

Dedicated hardware for video/image decoding, encoding, optical flow, post-processing

- **Capabilities**

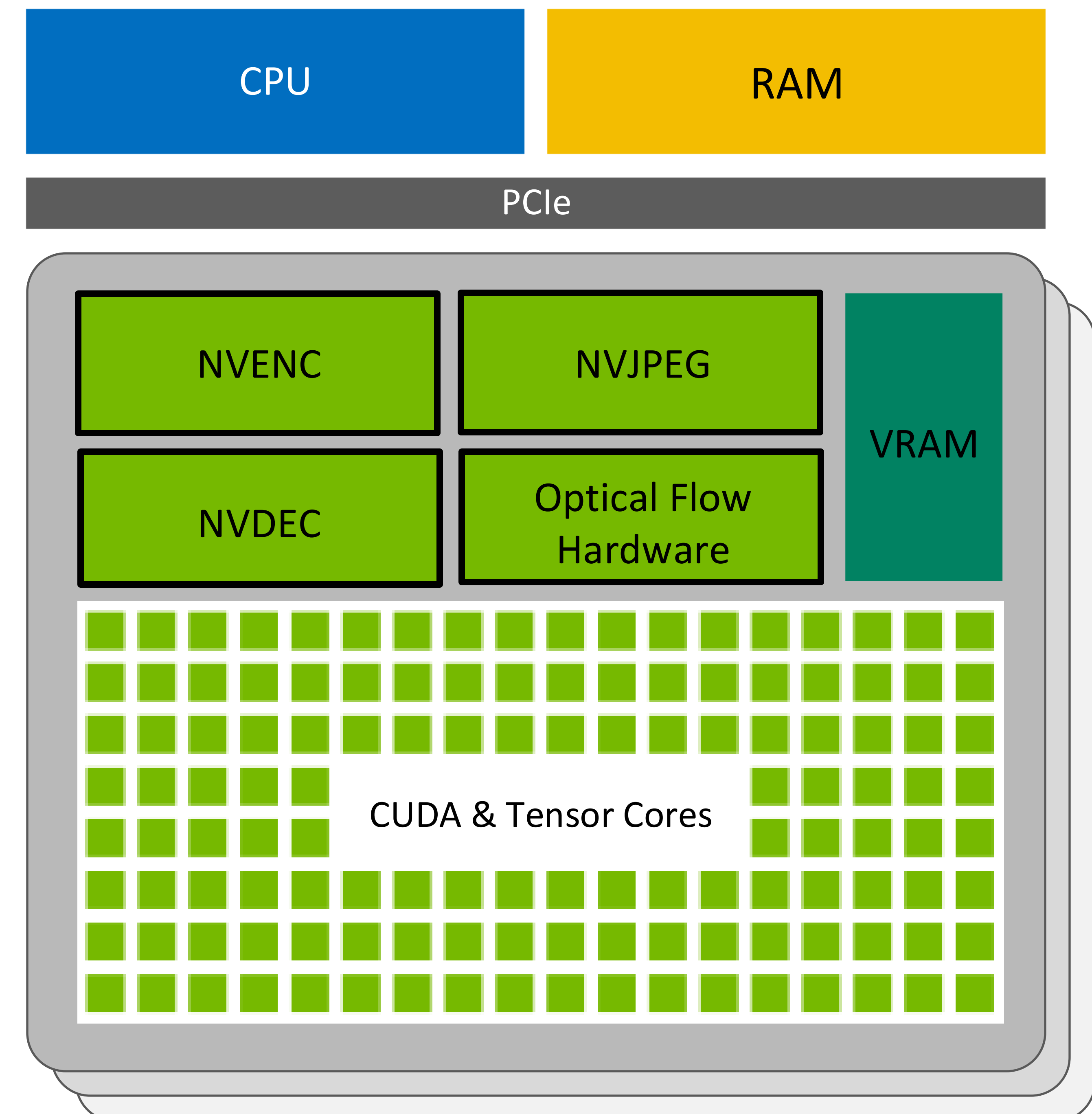
- NVENC – **Encode** video
- NVDEC – **Decode** video
- NVJPEG – **Decode JPEG** images
- Optical flow – **Track** pixels
- CUDA – **General-purpose compute, train, infer, ...**

- **Highly accelerated**

- **Power efficient**

- **Scalable**

- If you want to learn more <https://www.nvidia.com/en-us/on-demand/session/gtc25-s72756/>



Not all features are available in all GPUs. Please check NVIDIA developer zone web site for detailed information

How to exploit HW decoding?

DALI and PyNvVideoCodec

- Some NVIDIA libraries can help you with HW accelerated decoding
 - DALI for images and videos
 - PyNvVideoCodec for videos
- DALI and PyNvC are easy drop in replacements for existing code implementations
- What are the important things to remember
 - GPUs can be slowed down by CPU data preprocessing -> benchmark for your use case
 - CUDA zero copy is a great advantage -> decoding compressed data in GPU means less bandwidth for data transfer
 - Use memory friendly data formats -> few compressed files rather than millions images

Notes:

If interested in video, feel free to check out video materials I am collecting in a playbook on GitHub here, [accelerated-video-for-ai-playbook](#)

Multimodal data loading with Megatron Energon

What is Megatron Energon?

- Advanced multimodal dataloader for Megatron-LM
- Efficient loading and processing of diverse data
- Flexible configuration via Python API or CLI
 - `pip install megatron-energon`

Key Features

- **Multimodal Support:** Text, images, audio
- **Data Blending:** Mix datasets with fine-grained control
- **Distributed Loading:** Optimized for multi-node training environments
- **Save & Restore:** Resume training state from exact data position

Data Processing Capabilities

- **WebDataset Support:** Storage for multimodal data
- **Packing:** Optimize sequence length utilization
- **Grouping:** Smart batching of similar-length sequences
- **Joining:** Combine multiple dataset sources
- **Object storage:** Optimized loading from common object storage providers


Usage Example

```
from megatron.energon import get_train_dataset, get_loader,
    WorkerConfig

# load a training dataset and create a data loader
ds = get_train_dataset(
    '/my/dataset/path',
    batch_size=1,
    shuffle_buffer_size=100,
    max_samples_per_sequence=100,
    worker_config=WorkerConfig.default_worker_config(),
)

loader = get_loader(ds)
```


Code (PyNvC)

The background of the slide features a series of overlapping, diagonal, wavy bands in various shades of green, ranging from light lime to a darker forest green. These bands create a sense of depth and movement. On the far left, there is a solid, vertical green bar.

NVIDIA Academic Grant Program

Academic Grant Program

<https://www.nvidia.com/en-us/industries/higher-education-research/academic-grant-program/>

Benefits

- NVIDIA cloud, hardware, and/or software grants for research
- Access to NVIDIA models
- Letters of support for grant applications related to awarded projects
- Opportunities to present at GTC and network through NVIDIA's global channels
- Consideration for NVIDIA marketing support

Calls for Proposal

- [Gen AI: Training and Model Development](#)
- [Gen AI: Alignment and Inference](#)
- [Simulation and Modeling](#)

Apply at academicgrants.nvidia.com

Requirements

- Applicant must be a full-time faculty member at an accredited academic institution that awards research degrees to PhD students
- Proposal must adhere to the requirements included in the proposal template and relevant call for proposal
- Project must be accelerated with NVIDIA technology
- Submissions are accepted worldwide and on a rolling basis

NVIDIA Academic Grant Program

Accelerating Innovation in Academia

- **Program Benefits:**

- Receive NVIDIA cloud, hardware, and/or software grants for research
- Get access to NVIDIA models
- Receive letters of support for grant applications related to awarded projects
- Opportunity to present at GTC and network through NVIDIA's global channels
- Be considered for NVIDIA marketing support

- **Eligibility Requirements:**

- Applicant must be a full-time faculty member at an accredited academic institution that awards research degrees to PhD students
 - Postdocs and graduate students should work with a full-time faculty member to submit as a collaborator
- Proposal must adhere to the requirements included in the proposal template and relevant call for proposal
- Project must be accelerated with NVIDIA technology
- Submissions are accepted worldwide and on a rolling basis

- **Awardee Expectations:**

- Acknowledge NVIDIA Corporation in any resulting publications
- Provide an update every six months via a follow-up email questionnaire that asks for information about publications and presentations that NVIDIA's support enabled
- Submit proposals to present research outcomes (if any) at GTC or other NVIDIA events to promote the project

This is a competitive program. Not all projects that meet the eligibility requirements will be accepted.

NVIDIA's GenAI Libraries

Recap of NVIDIA's GenAI Training offerings

Core value Proposition

Nemo Framework: Easy to use OOTB FW with a large model collections for **Enterprise** users to experiment, train, and deploy.

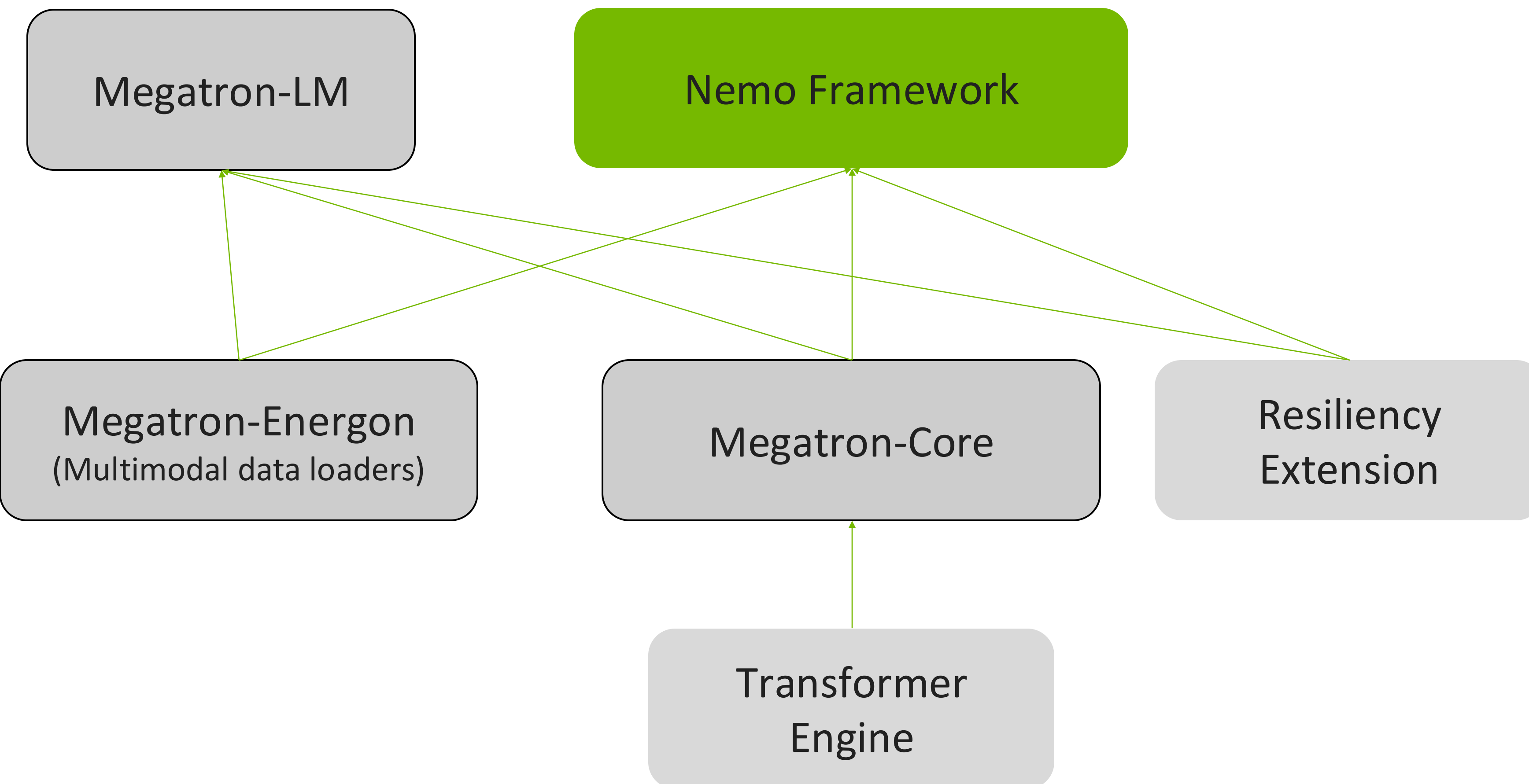
Megatron-LM: A lightweight reference training framework for using Megatron-Core to build your own LLM framework.

Megatron-Core: Library for GPU optimized techniques for training GenAI models at-scale.

Megatron-Energon: Multimodal data loaders for Megatron-Core.

Resiliency Extension: A library for resiliency features for PyTorch-based training

Transformer Engine: Accelerated kernels and FP8 mixed precision. Specific acceleration library, including FP8 on Hopper and Blackwell.



Key benefits of Megatron-Core

Performance at Scale

Parallelism Techniques

- Data/Tensor/Pipeline/Sequence/Context Parallel
- Virtual PP for improved performance
- Hierarchical Context Parallelism
- EP for MoE models
- Enc-Dec Parallelism (e.g. T5)

Memory Saving Techniques

- Selective Activation Recompute
- Offloading (Activation, weight, optimizer state)
- Attention: FAv2, FA-cuDNN, GQA, MQA, SWA
- SSM, MLA

Distributed Optimizers

- Zero-1 (fully implemented), Zero-2/3 (WIP)
- Precision aware optimizers (BF16, FP8)
- Custom FSDP - 15% speedup
- CPU offloading with hybrid device optimizers
- Overlapping CPU optimizer with data transfers

Additional Performance Features

- FP8 via Transformer Engine
- MLPerf Optimizations
- TRT-LLM based Inference
- Gradient accumulation fusion
- Pipeline comm optimizations
- Microbatch grouping for virtual pipeline stages

Mixture-of-Expert (MoE)

- Token drop and droptail approaches
- FP8 support
- Expert MP with configurable expert TP/MP
- Expert DP
- Grouped GEMM for MoE layers
- All-to-all token dispatche

Customizable Building Blocks

Optimized transformer blocks with **modular and composable APIs**

Canonical architectures:

- Decoder (GPT), Encoder (BERT), Enc-Dec (T5)
- RAG (RETRO), MoE, ViT/DiT
- Hybrid (Mamba SSM)

Releases with SOTA features

Multimodal Training

- Sequence and context parallelism for LLaVA
- Variable sequence lengths across microbatches

Blackwell support

- QAT for FP4 inference
- MXFP8 recipe support

Leverage Cutting-edge Research

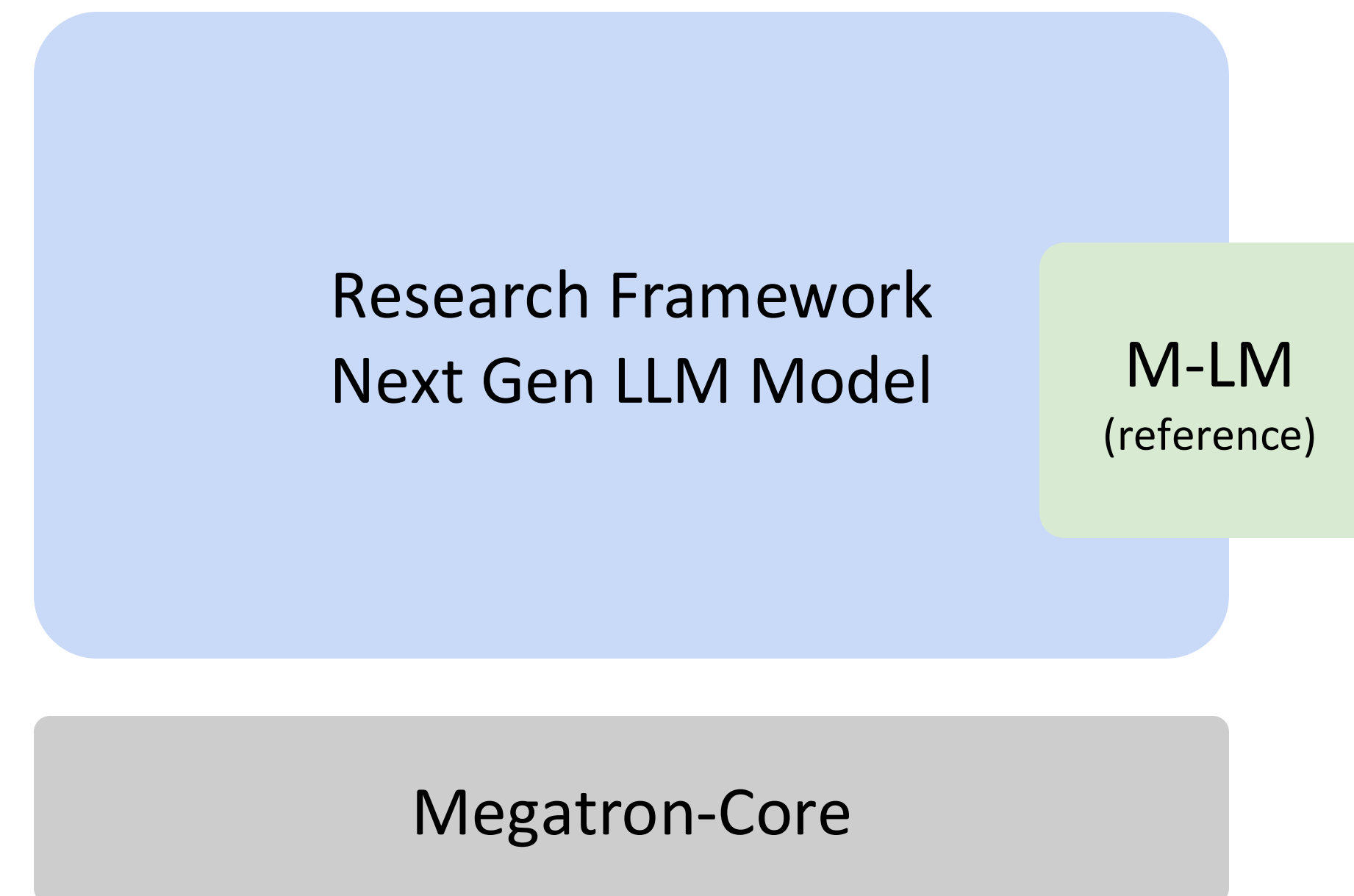
- Stay at the forefront of distributed training
- SSM-based hybrid models

Scalability & Training Resiliency

- Fast distributed checkpointing
- Integration with NVIDIA Resiliency Extension (WIP)
 - Hang, Straggler and SDC detection
 - In-job, In-process restart
 - Hierarchical Checkpointing
- Configurable distributed timeout
- NCCL comm configuration
- Gloo process groups for CPU operations

Software Choices for LLM Developers

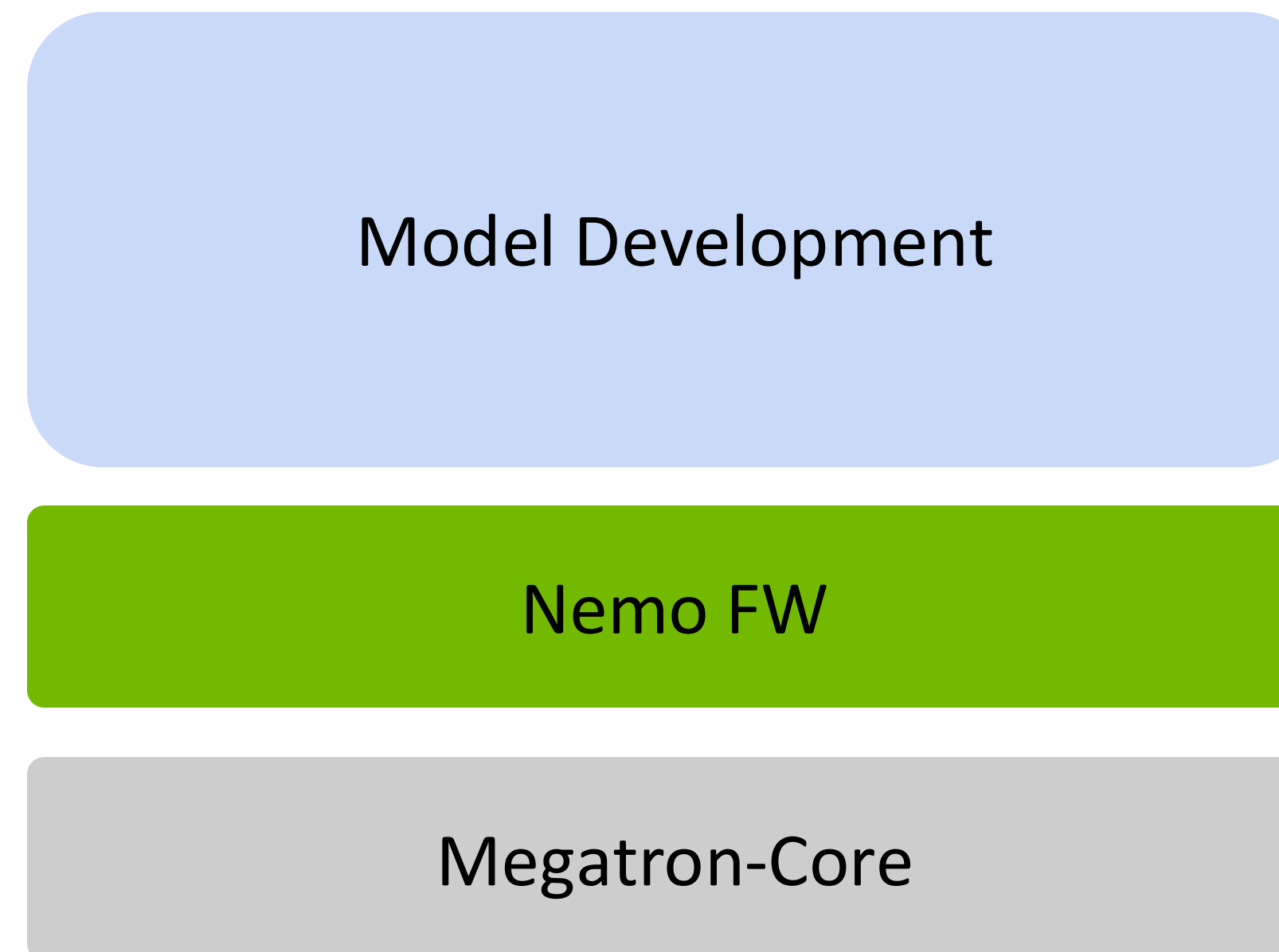
Persona 1: Research in LLM Framework & Models



PyTorch

Core optimizations/kernels for LLM training at scale with latest updates from NV.

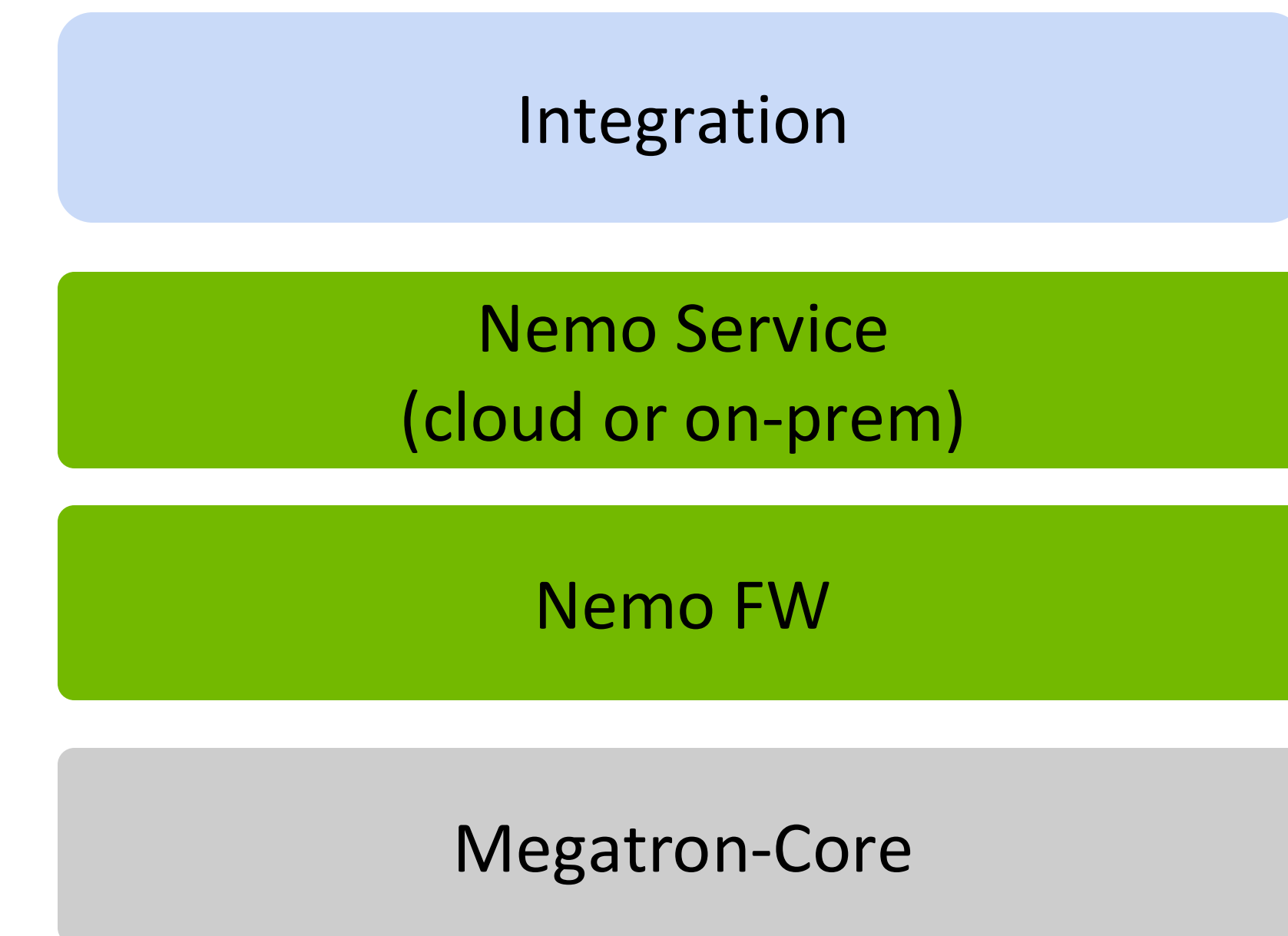
Persona 2: Develop your own LLM and ConvAI models from scratch



PyTorch+Lightning

End-to-end training open source framework.
Train from scratch w/guaranteed convergence on a specific set of SotA model architectures and data types
Fine-tuning customization techniques
Optimized conversion to TRT

Persona 3: Deploy and operationalize SOTA models for production



PyTorch+Lightning

Deploy and tune LLM to production: Fine tune, containerize, microservices, enterprise integration, RAG
Pre-built containers and services to operate infrastructure and MLOps integration

Value

Challenges

Requires developer to have their own framework implementation.
Only for experts in the field of distributed AI training software.

Expects AI practitioner skills (training scripts, job).
User provides infra and operates infra.
Traditional framework only, no automation/services/MLOps

Supports only pre-trained community and NV specific models.
No train from scratch or novel model architectures
Microservice interfaces, not a framework

