Scaling and accelerating LLM trainings

Andrea Pilzer, Solutions Architect | NHR PerfLab Seminar/15.01.26

Agenda

- Intro & Motivational Example
- TL;DR Parallelization Techniques
- Parallelization Techniques

Andrea Pilzer

apilzer@nvidia.com

- Since May 2022 Solution Architect at NVIDIA
  - SA HER, Leading NVAITC Italy
  - VLMs, Video Models, LLMs
- 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

NVIDIA's Evolution From Chips to an AI Infrastructure Company

ACCELERATED COMPUTING

= ~100X SPEED-UP

~3X POWER

~1.5X COST

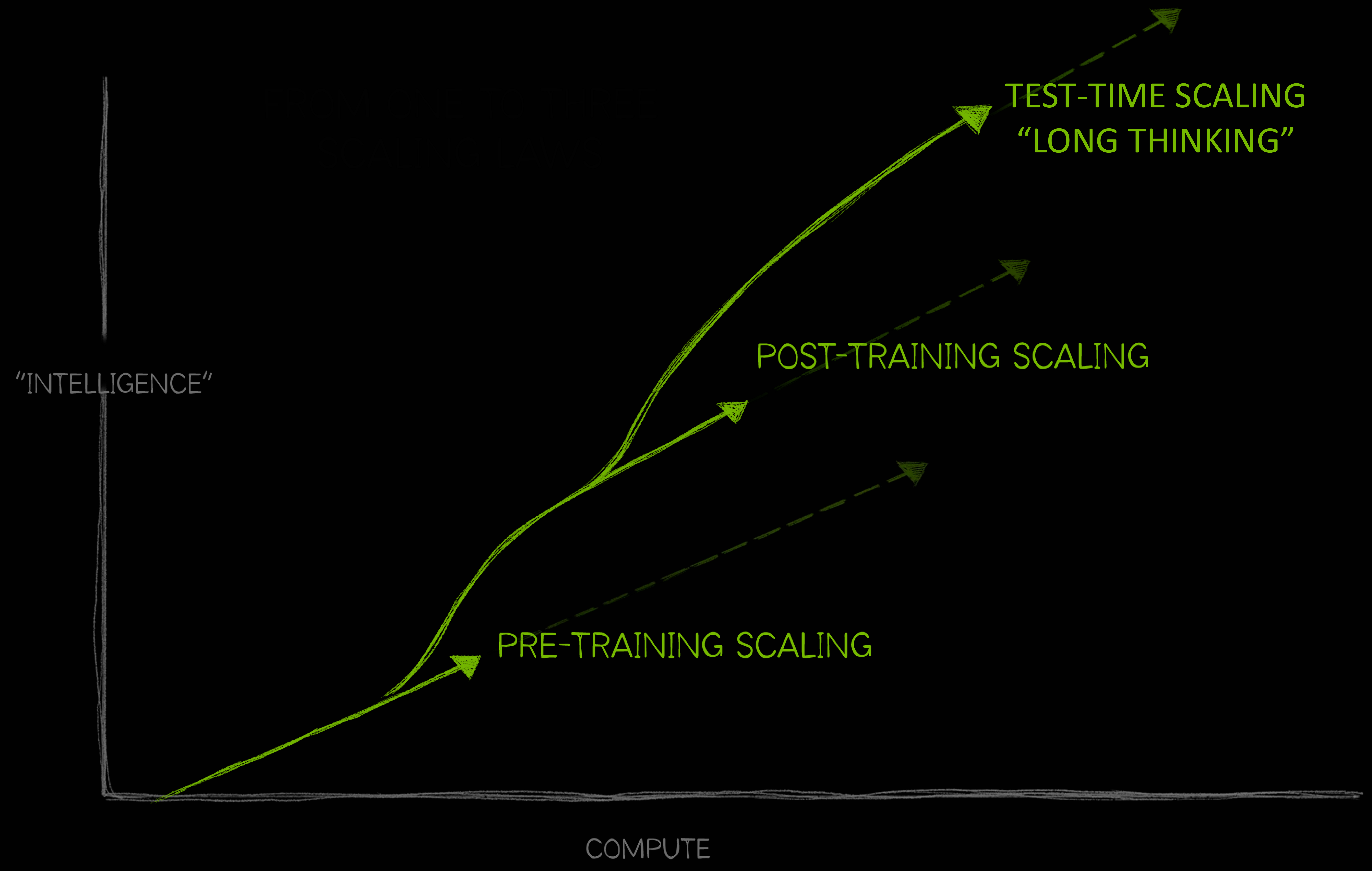
60X PERF / \$ OR 98% SAVINGS

30X PERF / W OR 97% SAVINGS

"THE MORE YOU BUY... THE MORE YOU SAVE"

AI Scaling Laws Drive Exponential Demand for Compute

New “long thinking” supercharges inference scaling



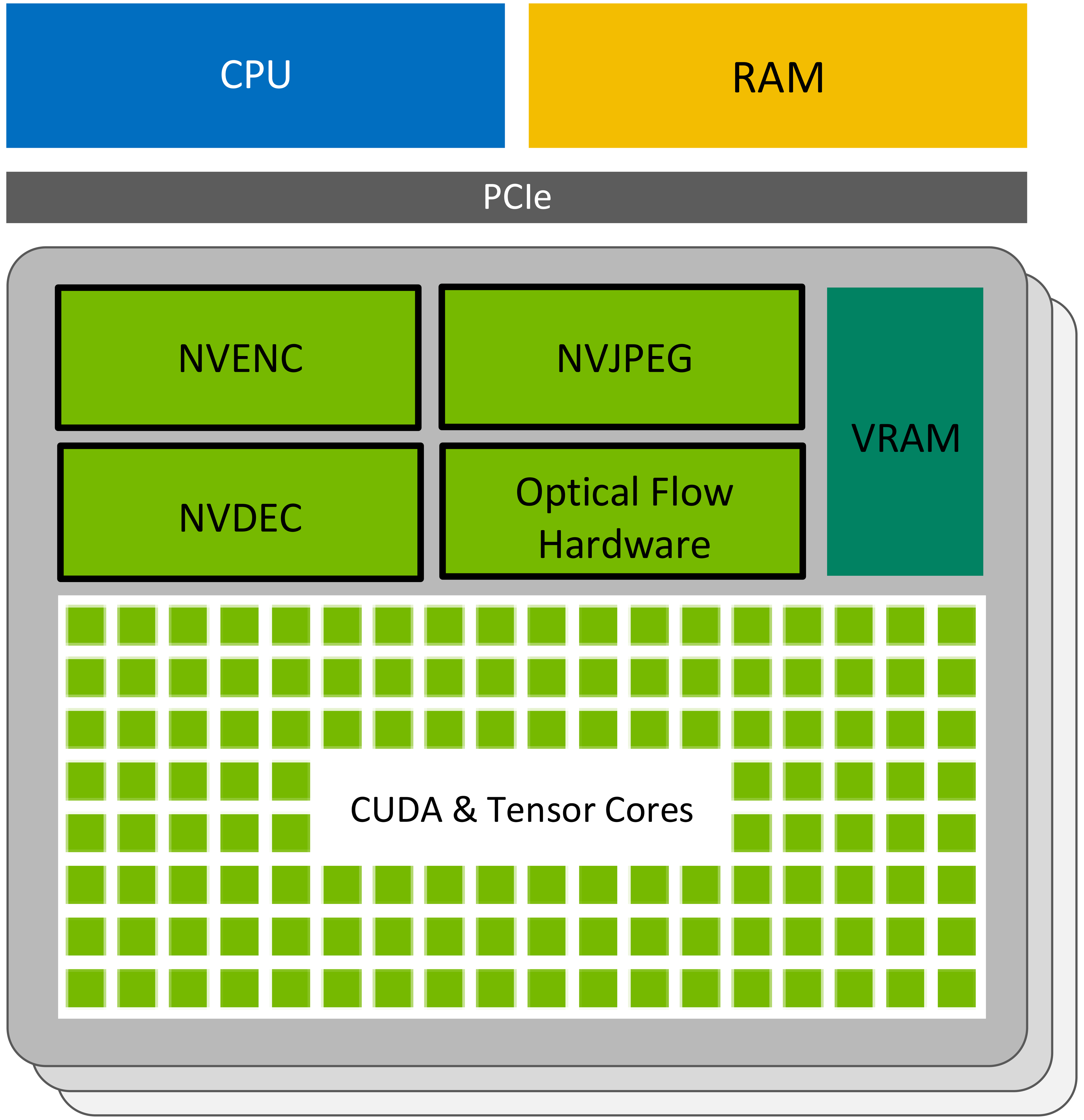
Demo Leonardo

TL;DR

This Session in One Slide (1)

NVIDIA GPUs: Remember you are using HW

- **Capabilities**
  - Tensor Cores – accelerate GEMM
  - NVENC – **Encode** video
  - NVDEC – **Decode** video
  - NVJPEG – **Decode JPEG** images
  - Optical flow – **Track** pixels
  - CUDA – **General-purpose compute, train, infer**, …
  - vRAM – our OOM friend ☺ (the most expensive part)
- **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

This Session in One Slide (2)

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
Distributed Optimizer*	No	Yes
Offloading	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

[3] Efficient Training on Multiple GPUs

This Session in One Slide (3)

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

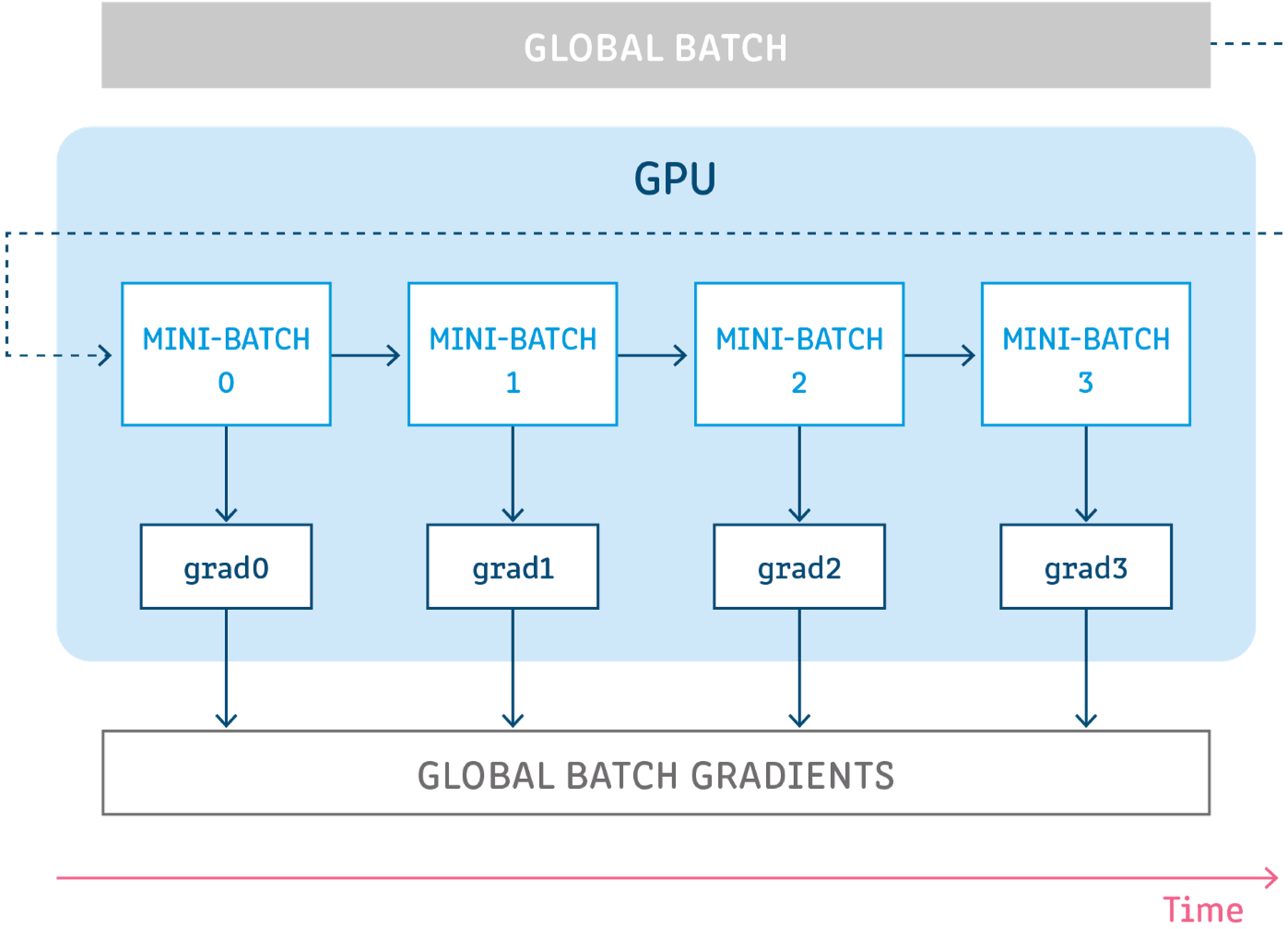
Parallelization Techniques

Utilizing a single GPU efficiently

Dealing with memory constraints

Gradient accumulation

- Gradient accumulation is a mechanism to split the batch of samples — used for training a neural network — into several mini-batches of samples that will be run sequentially.



https://towardsdatascience.com/what-is-gradient-accumulation-in-deep-learning-ec034122cfa

Gradient accumulation

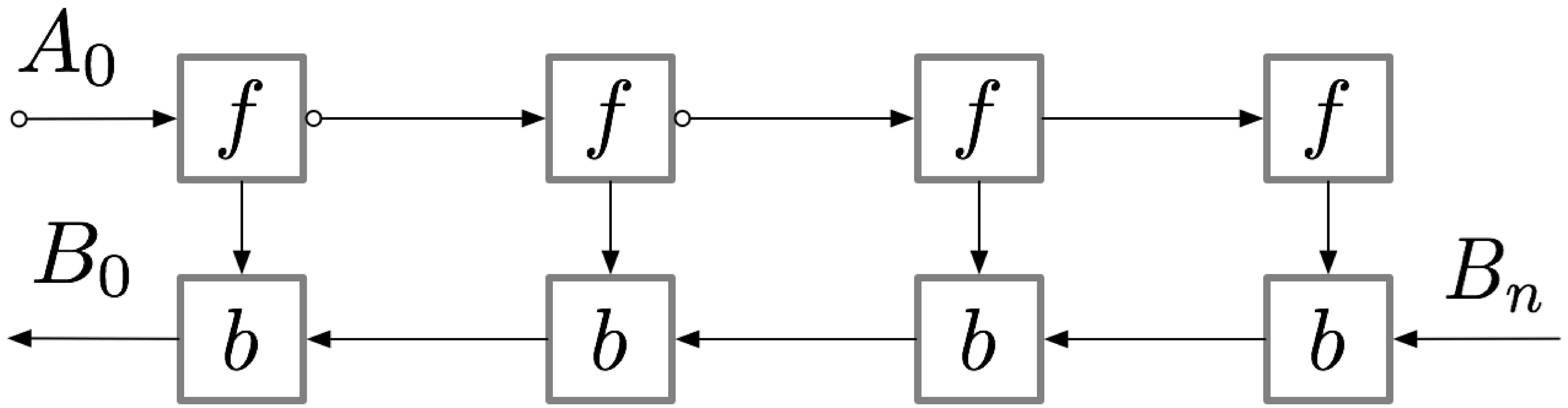
```
optimizer = ...

for epoch in range(...):
    for i, sample in enumerate(dataloader):
        inputs, labels = sample
        optimizer.zero_grad()
        # Forward Pass
        outputs = model(inputs)
        # Compute Loss and Perform Back-propagation

        loss = loss_fn(outputs, labels)
        loss.backward()
        # Update Optimizer          optimizer.step()
```

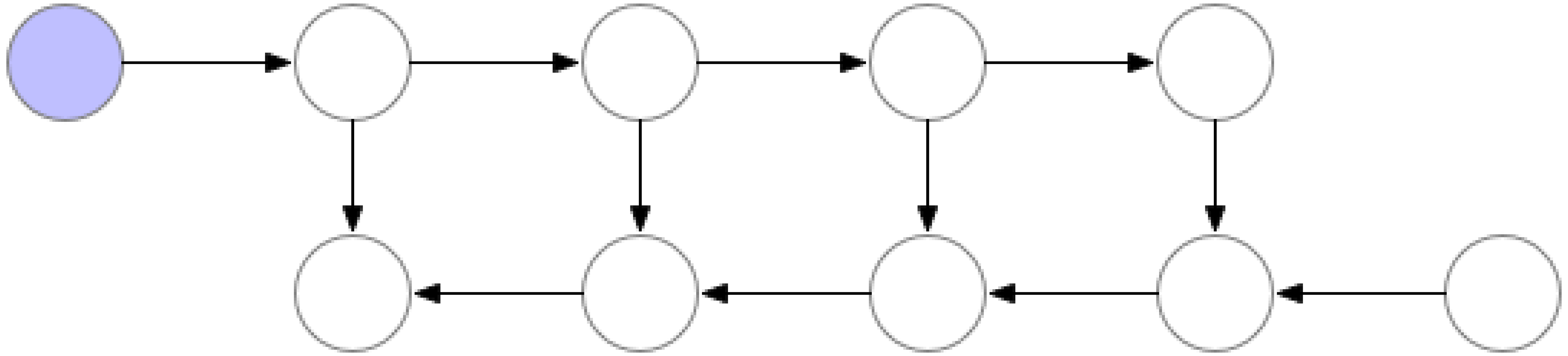
```
optimizer = ...
NUM_ACCUMULATION_STEPS = ...
for epoch in range(...):
        for idx, sample in enumerate(dataloader):
        inputs, labels = sample
        # Forward Pass
        outputs = model(inputs)
        # Compute Loss and Perform Back-    propagation
        loss = loss_fn(outputs, labels)
        # Normalize the Gradients
        loss = loss / NUM_ACCUMULATION_STEPS
        loss.backward()
        if ((idx + 1) % NUM_ACCUMULATION_STEPS ==
            0) or (idx + 1 == len(dataloader)):
                optimizer.zero_grad()
                # Update Optimizer
                optimizer.step()
```

Activation Re-computation or gradient checkpointing

https://pytorch.org/docs/stable/checkpoint.html

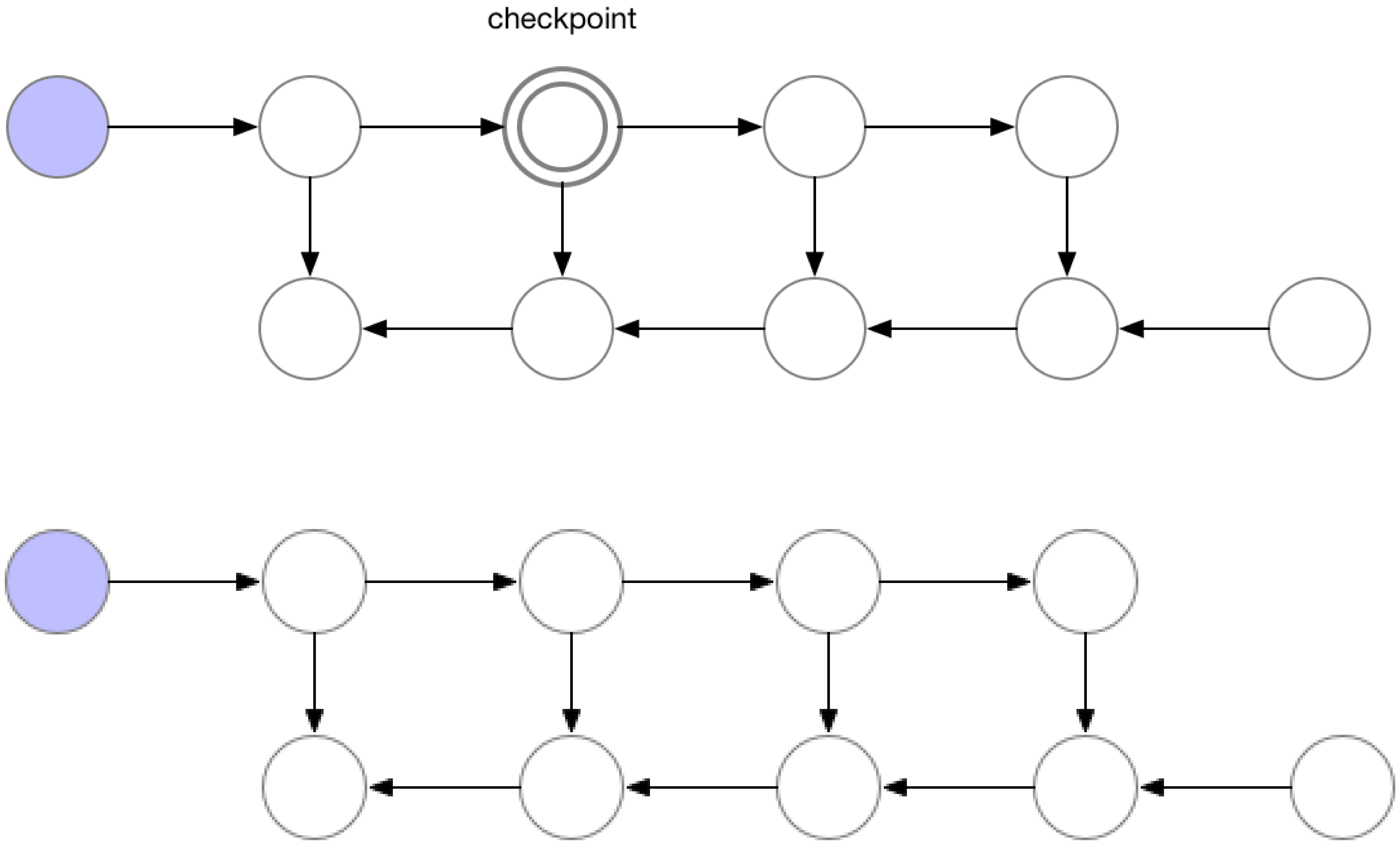
- The memory intensive part of training deep neural networks is computing the gradient of the loss by backpropagation.
- By checkpointing nodes in the computation graph defined by your model, and recomputing the parts of the graph in between those nodes during backpropagation, it is possible to calculate gradients at reduced memory cost.





https://github.com/cybertronai/gradient-checkpointing

Activation Re-computation or gradient checkpointing



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

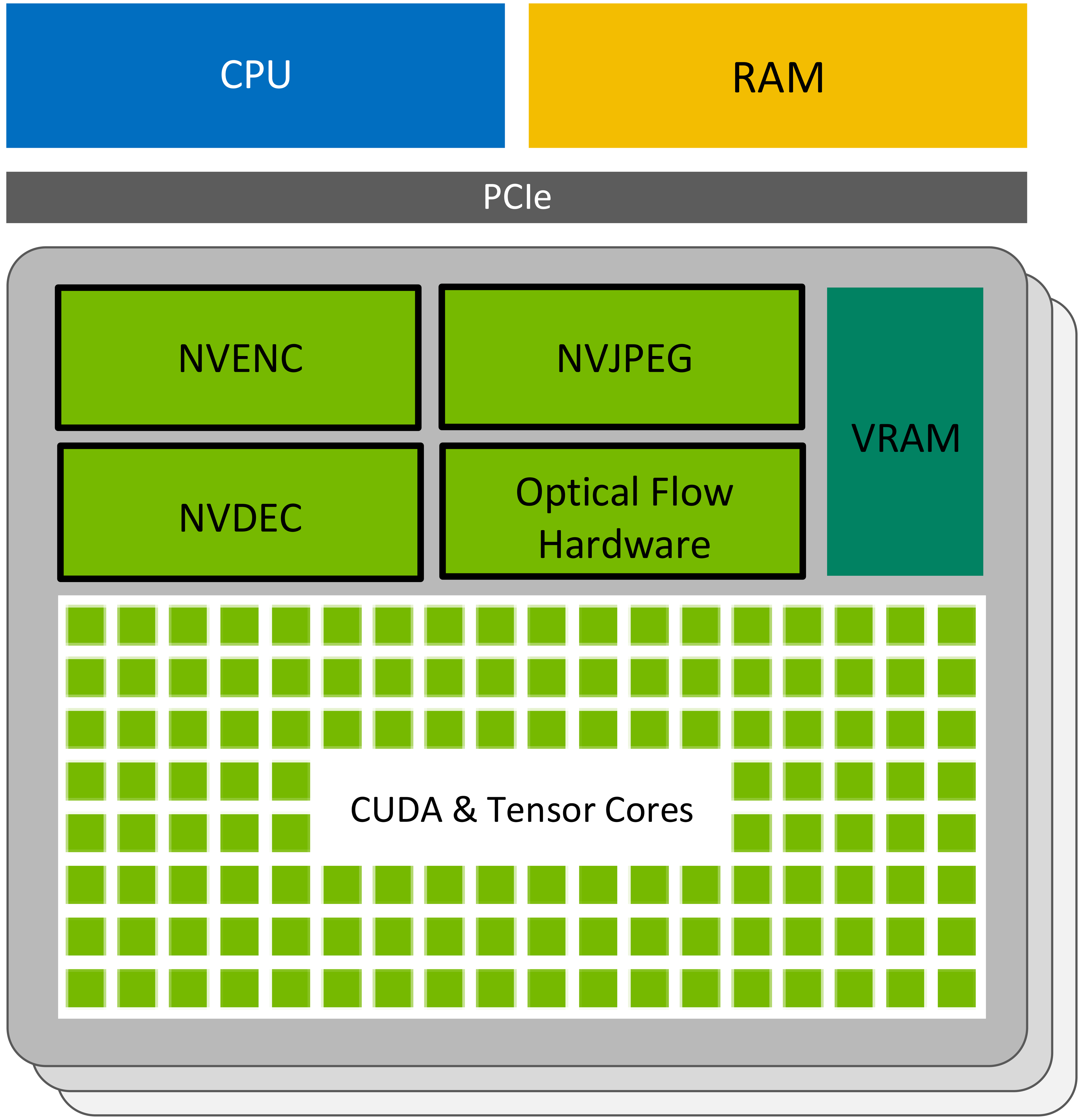
- Asyncronous copy, CUDA can help in hiding data moving cost behind other operations
  - data = data.to(device) ⇒ data = data.to(device, **non_blocking=True**)

- 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 bandwith 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`

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

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

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)
```

[Megatron Energon Docs]()

Code (PyNvC)

Data Parallelism

(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 it's 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

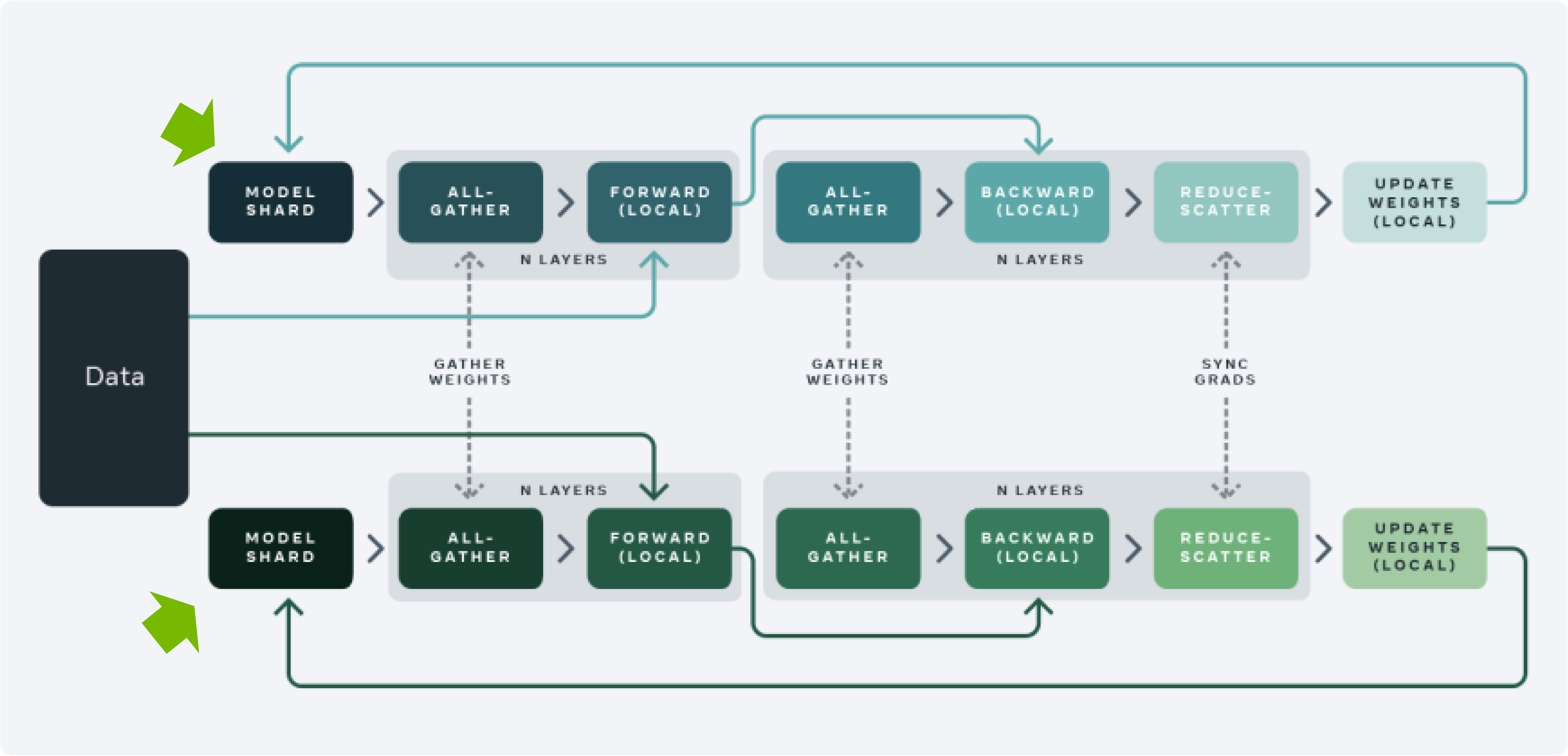
Parameters Sharding

Distributed Data Parallel - DDP

FairScale: Fully Sharded Data Parallel - FSDP

For each GPU:

1. Get the shard of the model
2. Get the shard of the data
3. Local forward pass: **Gather weights** from the others
4. Local backward pass: **Gather again** weights from the others
5. Local weights shard update: *Synchronize* Gradients



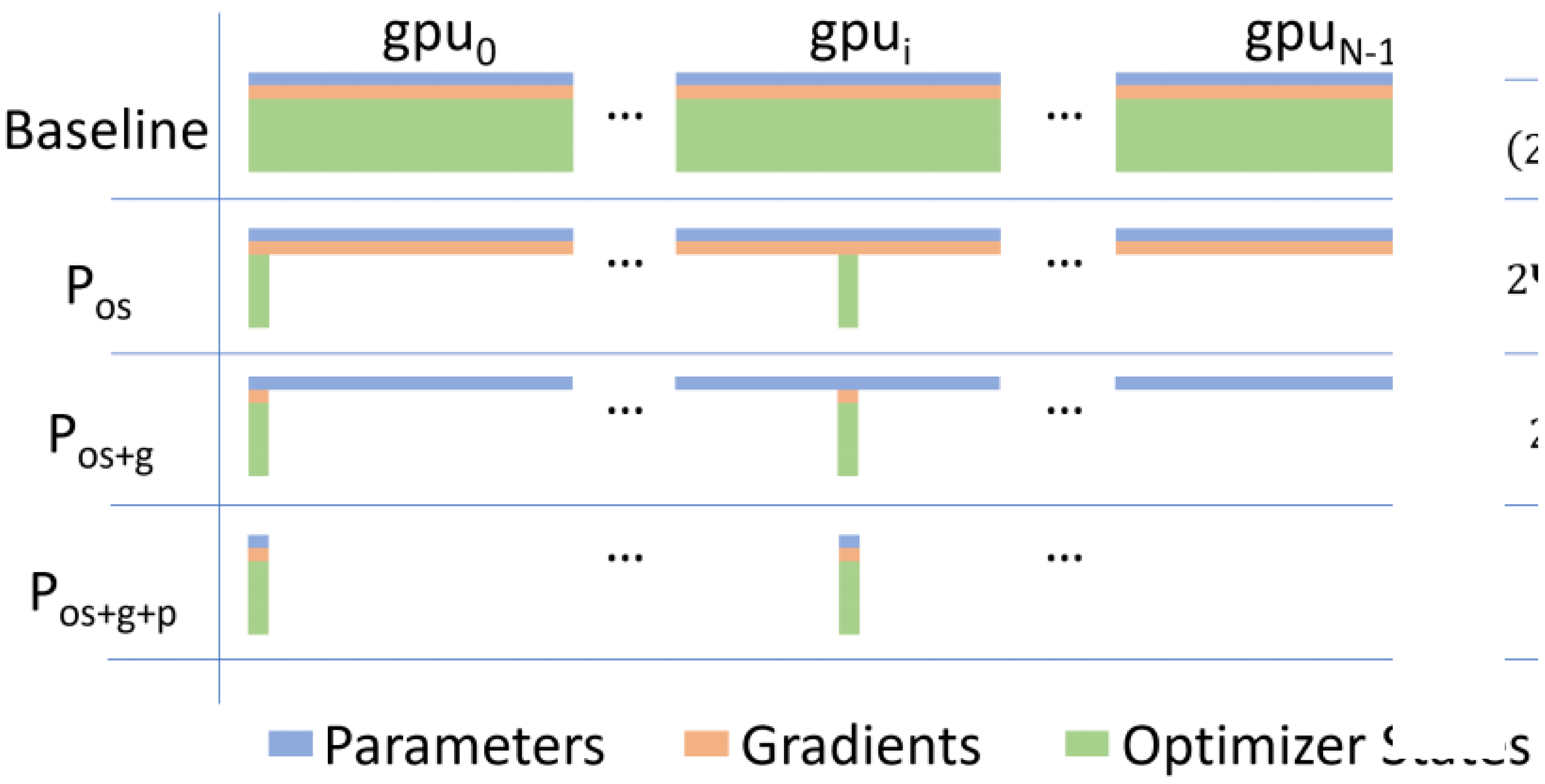
https://fairscale.readthedocs.io/en/stable/api/nn/fsdp.html

Sharded Data Parallelism

ZeRO: Zero Redundancy Optimizer

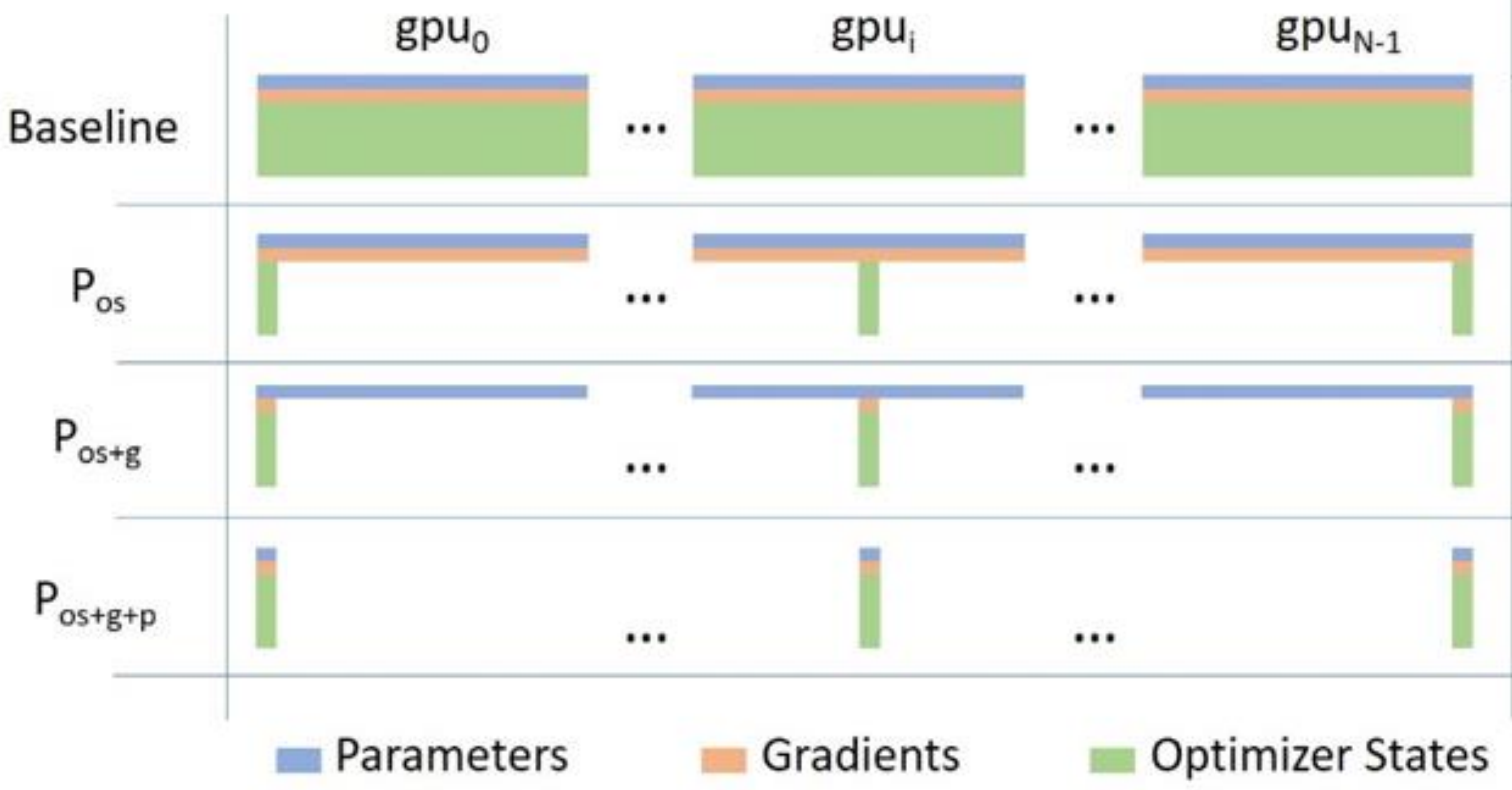
deepspeed

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



Sharded Data Parallelism

Communication overheads

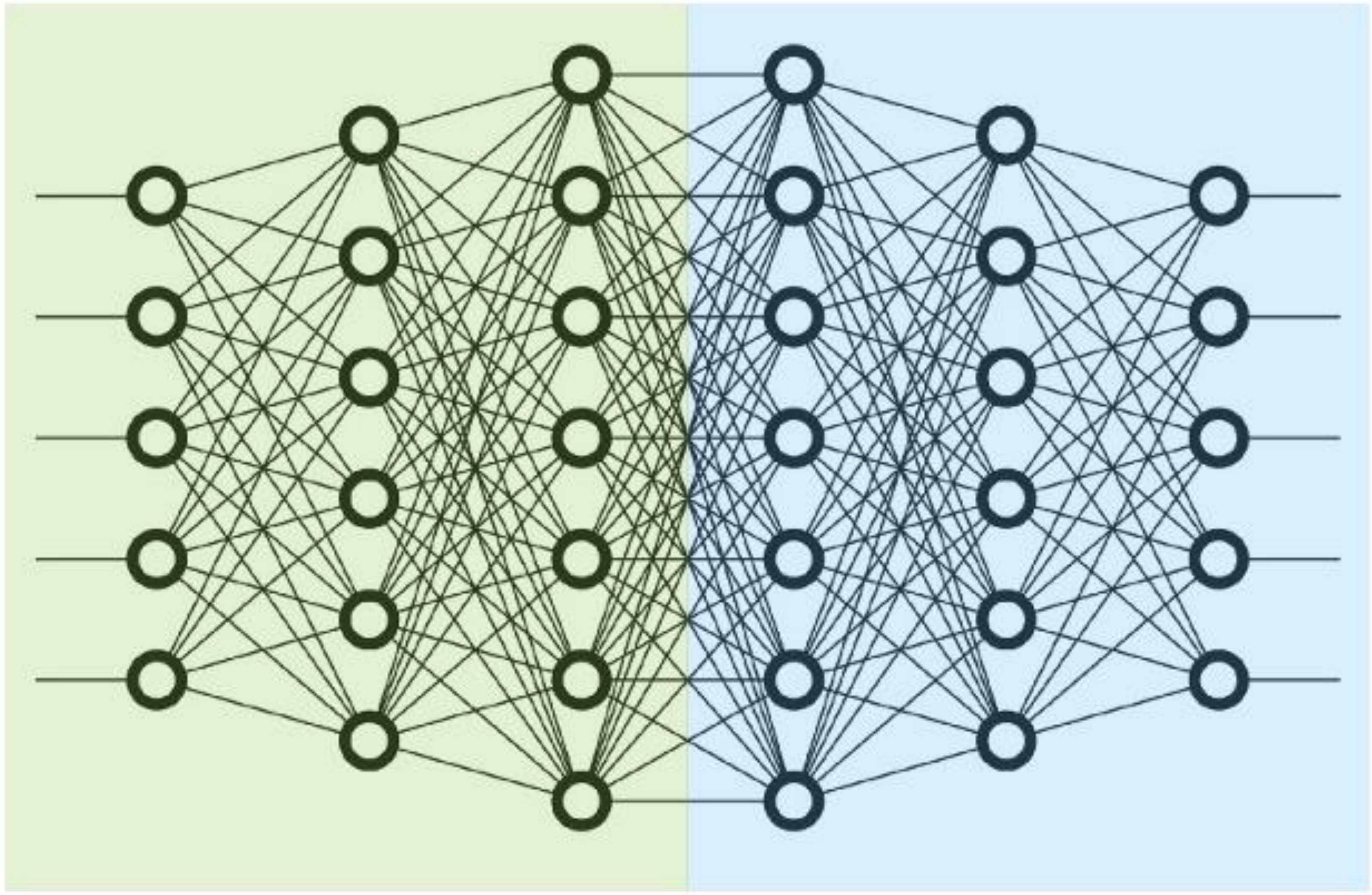


	Memory Consumption		Comm Volume
	Formulation	Specific Example K=12 Ψ=7.5B N_d=64	
Baseline	$(2+2+K)*\Psi$	120GB	1x
P_{os}	$2\Psi+2\Psi+\frac{K*\Psi}{N_d}$	31.4GB	1x
P_{os+g}	$2\Psi+\frac{(2+K)*\Psi}{N_d}$	16.6GB	1x
P_{os+g+p}	$\frac{(2+2+K)*\Psi}{N_d}$	1.9GB	1.5x

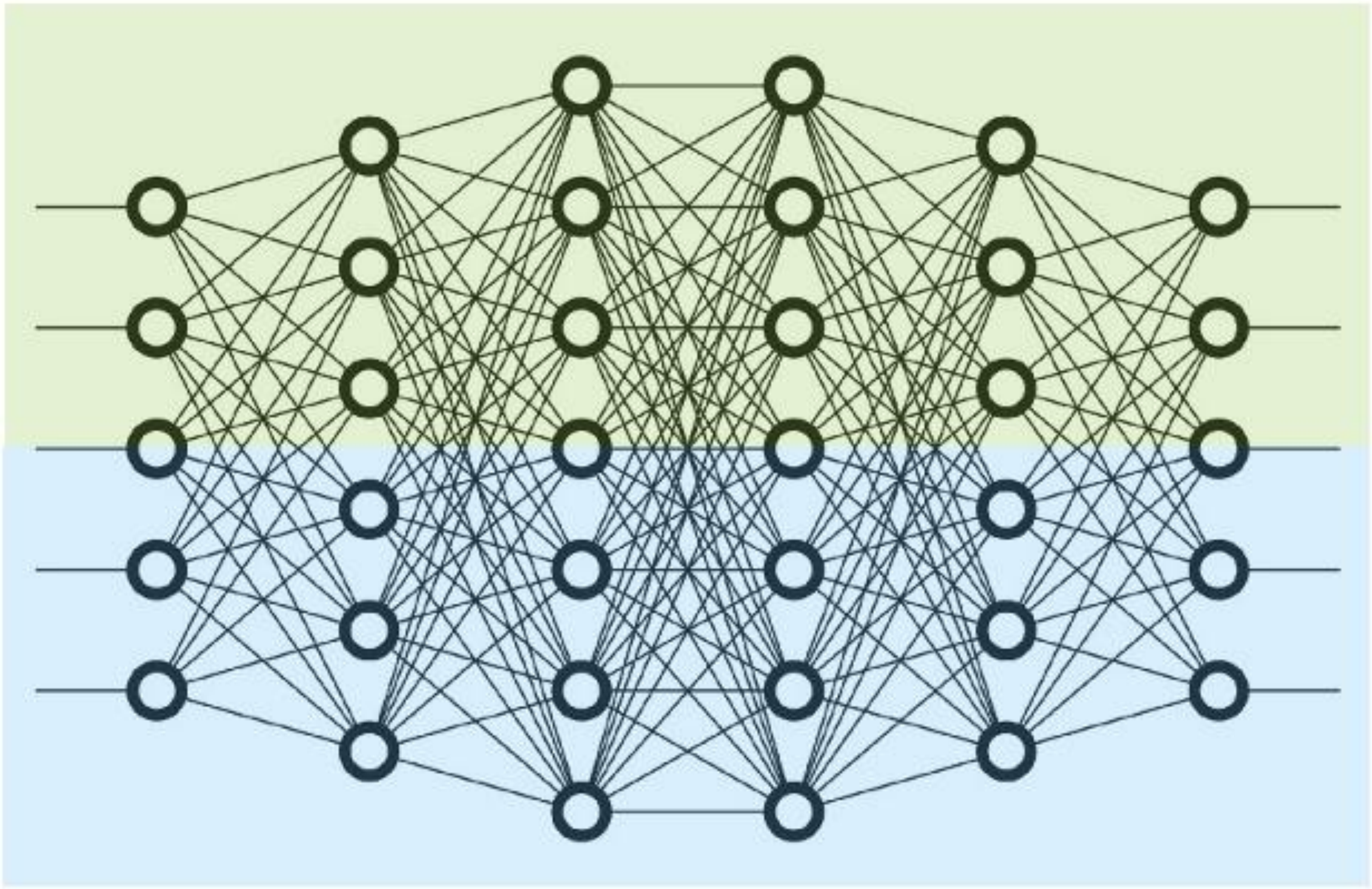
Model Parallelism

Model Parallelism

- **Pipeline (Inter-Layer) Parallelism**
  - Split sets of layers across multiple devices
  - Layer 0,1,2 and layer 3,4,5 are on difference devices



- **Tensor (Intra-Layer) Parallelism**
  - Split individual layers across multiple devices
  - Both devices compute difference parts of Layer 0,1,2,3,4,5



NVIDIA

Pipeline Parallelism

Pipeline Parallelism

Challenges

Pipeline Parallelism
Challenges – Idle Workers

Pipeline Parallelism

Split batch into micro batches and pipeline execution

Pipeline Parallelism

Split batch into micro batches and pipeline execution

Pipeline Parallelism

Split batch into micro batches and pipeline execution

NVIDIA.

Pipeline Parallelism

Split batch into micro batches and pipeline execution

Pipeline Parallelism

Split batch into micro batches and pipeline execution

Pipeline Parallelism

Split batch into micro batches and pipeline execution

Pipeline Parallelism

Split batch into micro batches and pipeline execution

$$\text{total time} = (m + p - 1) \times (t_f + t_b)$$
$$\text{ideal time} = m \times (t_f + t_b)$$
$$\text{bubble time} = (p - 1) \times (t_f + t_b)$$

$$\text{bubble time overhead} = \frac{\text{bubble time}}{\text{ideal time}} = \frac{p-1}{m}$$

p : number of pipeline stages
m : number of micro batches
t_f : forward step time
t_b : backward step time

Forward pass

Backward pass

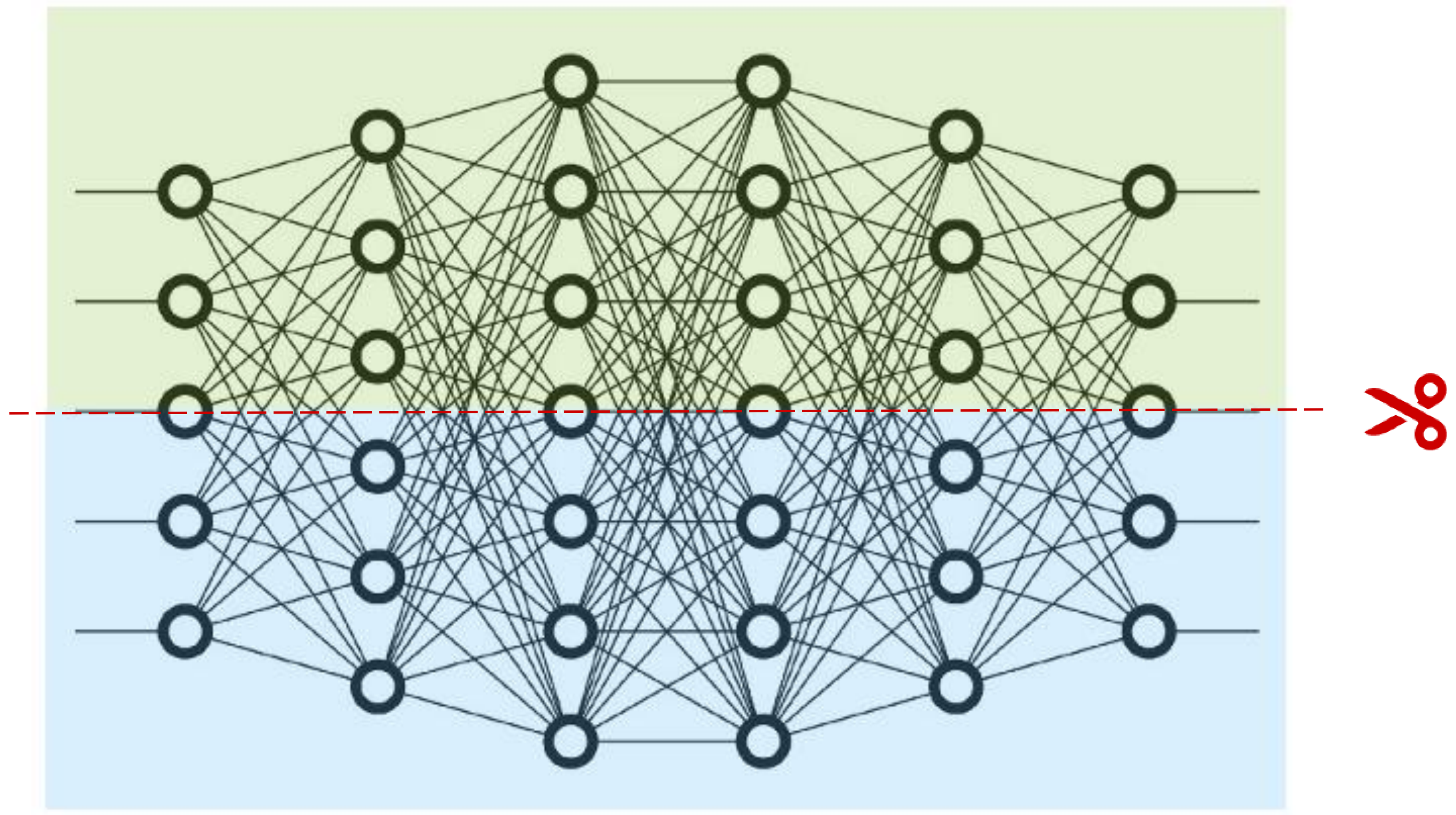
NVIDIA

Code (DualPipe)

Tensor Parallelism

Tensor Parallelism

- Relatively simple to implement
- Easier to load-balance
- Less restrictive on the batch-size (avoids bubble issue in pipelining)
- Tensor parallelism works well for large matrices
- Example: Transformers have large GEMMs



Simple example of Tensor parallelism

Row Parallel Linear Layer

Column Parallel Linear Layer

NVIDIA